VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky

BAKALÁŘSKÁ PRÁCE

2014 Jan Lysoň

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Možnosti OpenGL
Capabilities of OpenGL

2014 Jan Lysoň

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jan Lysoň**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Možnosti OpenGL
Capabilities of OpenGL

Zásady pro vypracování:

Grafické rozhraní OpenGL je dnes již standard pro počítačovou grafiku a zejména pro vykreslování 3D. Cílem této práce je zaměřit se zejména na OpenGL verze 4.x a popsat novinky přicházející s touto verzí.

1. Nastudujte problematiku OpenGL.
2. Popište a na ukázkových příkladech demonstrujte novinky OpenGL verze 4.x. Příklady vytvořte tak, aby bylo možné testovat a vyhodnocovat jejich přínos a aby je bylo možné použít ve výuce jako ukázkové příklady.
3. Vše pečlivě zdokumentujte, aby bylo možné v práci pokračovat.

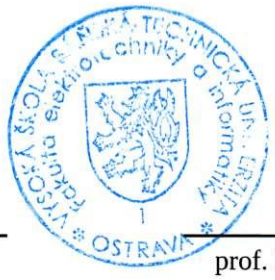
Seznam doporučené odborné literatury:

[1] D. Shreiner, G. Sellers, J.M. Kessenich and B. M. Licea-Kane, OpenGL Programming Guide, 8th Edition. 2013. 984 p. ISBN 978-0-321-77303-6
[2] http://www.opengl.org/

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Martin Němec, Ph.D.**

Datum zadání: 01.09.2013
Datum odevzdání: 07.05.2014



doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Tímto bych rád poděkoval svému vedoucímu, Ing. Martinu Němcovi, Ph.D., za odborné rady a jeho pozitivní přístup během konzultací k této práci.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Dne 30. 4. 2014 Podpis: Jan Lyson

Abstrakt

Z důvodu rychlého vývoje grafických karet je OpenGL, které je dnes považováno za jeden z grafických standardů, nuceno se neustále přizpůsobovat jejich možnostem a tím měnit i svou specifikaci. V této práci jsou zpracována vybraná témata z oblasti vizualizace dat, na kterých jsou demonstrovány možnosti moderního OpenGL verze 4, včetně hlavních novinek této verze. Pro každé jednotlivé téma v této práci je obsažen teoretický úvod s možným doplněním základních informací o OpenGL funkcionalitě, která je využita v praktické ukázce. Praktická ukázka pro každé téma obsahuje jednu nebo více ukázkových aplikací, přičemž jsou popsány hlavní části implementace jedné z nich. Ukázkové aplikace jsou vytvořeny s pomocí programovacích jazyků C++ a GLSL, knihovny OpenGL a dalších potřebných knihoven.

Klíčová slova

vizualizace dat, OpenGL, GLSL, objekty, textury, osvětlení, Phongův osvětlovací model, normal mapping, stíny, animace, částicový systém, teselace

Abstract

The OpenGL which is concerned as one of graphic standards today is forced to constantly adapt to capabilities of graphic cards and change a specification due to rapid development of them. In this thesis are processed chosen themes from the field of data visualization which are used to demonstrate capabilities of modern OpenGL version 4 including the main news of this version. For every single theme in this thesis is contained theoretical introduction with possible addition of basic information about OpenGL functionality which is used in practical demonstration. Practical demonstration contains one or more demonstration applications with description of main parts for one of them. Demonstration applications are created using programming languages C++ and GLSL, library OpenGL and other necessary libraries.

Key words

data visualization, OpenGL, GLSL, objects, textures, lighting, Phong lighting model, normal mapping, shadows, animation, particle system, tessellation

Obsah

Kapitola 1

Úvod

Počítačová grafika je část informatiky, která se zabývá zejména 2D a 3D grafikou. Hlavním účelem počítačové grafiky je zpracování a vizualizace dat, která v dnešní době zasahuje do téměř všech oblastí, kde pro některé je používání počítačové grafiky nezbytnou součástí pro jejich provoz. Příkladem těchto oblastí jsou různá průmyslová odvětví, jako je automobilový průmysl, hutní průmysl a mnoho dalších, kde je počítačové grafiky využíváno například pro vizualizaci výrobních procesů. Široké využití je zejména v oblasti lékařství pro vizualizaci dat z různých diagnostických metod, například dat získaných ze skenování pomocí počítačové tomografie (angl. Computed Tomography, zkr. CT) aj. Další využití je potom pro tvorbu různých grafických softwarů, jako jsou například CAD programy.

1.1 Grafické standardy

V dnešní době existují dva hlavní grafické standardy pro vizualizaci dat, které si vzájemně konkurují i přesto, že jejich možnosti vizualizace jsou téměř ekvivalentní. Jedná se o standardy OpenGL a DirectX. Oba standardy nabízejí funkcionalitu pro vykreslování 2D i 3D grafiky, nicméně v případě OpenGL se jedná o standard, který pouze specifikuje programové rozhraní (angl. Application programmable interface, zkr. API), jehož implementace je poté závislá na výrobci grafického hardwaru, oproti tomu DirectX je standard, který spravuje a pro který poskytuje implementaci, pouze firma Microsoft. Oba standardy nabývají určitých výhod, které hrají významnou roli při výběru mezi nimi. [1][2]

Patrnou výhodou DirectX je fakt, že je tvořen souborem knihoven, které implementují programové rozhraní jak pro vykreslování 2D a 3D grafiky (Direct2D, Direct3D), tak i pro další užitečné věci pro tvorbu grafické aplikace, jako je například rozhraní pro přehrávání zvuku (DirectSound), vykreslování různých fontů (DirectWrite) nebo pro detekci vstupů z klávesnice, myši nebo ovladače (DirectInput). Při použití DirectX je tedy vetšina potřebných nástrojů pohromadě. [1]

Hlavní výhodou OpenGL je otevřenost tohoto standardu, což znamená, že výrobci, kteří implementují OpenGL rozhraní mají možnost přidat dodatečné implementace, například pro využití nových možností grafického hardwaru, v podobě rozšíření (angl. extension). Tato rozšíření nemusí být obsažena v OpenGL implementaci ostatních grafických karet, nicméně určitá rozšíření mohou být poté schválena jako povinná součást OpenGL implementace. Další výhodou OpenGL

je jeho multiplatformnost, která umožňuje přenositelnost aplikací mezi majoritními operačními systémy, v čemž se liší od DirectX, který je možné využívat pouze s operačním systémem Microsoft Windows nebo se systémem pro konzole XBOX. [1][2]

1.2 Zaměření a struktura práce

V této práci je pro vizualizaci dat využit standard OpenGL, který byl vybrán převážně pro svou multiplatformnost. Určité kapitoly jsou zaměřeny na vybrané možnosti využití OpenGL standardu, pro které jsou rozebrány a popsány hlavní části způsobu implementace. V příloze této práce jsou poté obsaženy konečné a spustitelné aplikace jako ukázkové příklady. Stejně tak i podrobnější teoretická část pro určité kapitoly je obsažena v příloze, na kterou bude v textu odkazováno.

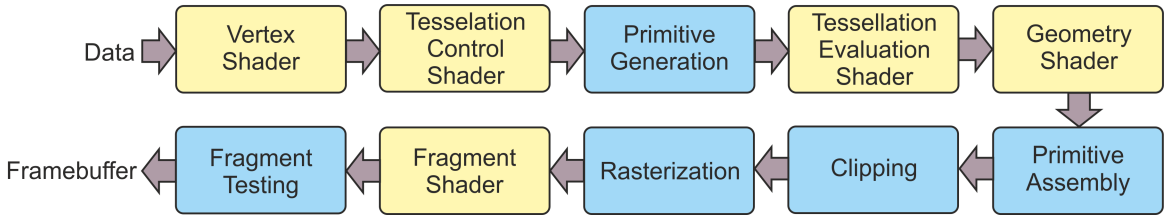
Kapitola 2

Open Graphics Library

Open Graphics Library (zkr. OpenGL) je multiplatformní specifikace programového rozhraní pro tvorbu 2D a 3D počítačové grafiky, navržena americkou firmou Silicon Graphics Inc. (zkr. SGI), která se zabývala výkonnými výpočetními systémy, včetně hardwaru a softwaru, původně zaměřena na zobrazovací terminály 3D grafiky. Předchůdcem dnešního standardu OpenGL je grafické API nazývané IRIS GL (Integrated Raster Imaging System Graphics Library), které sloužilo pro vytváření 2D a 3D grafiky na pracovních stanicích firmy SGI. Později však, z důvodu složité údržby zdrojového kódu, byl kód IRIS GL upraven a zpřehledněn, což dalo, v roce 1992, za vznik OpenGL standardu. Společně se vznikem OpenGL bylo založeno nezávislé konsorcium s názvem OpenGL ARB (Architecture Review Board), které mělo za úkol spravovat a také schvalovat úpravy a nové vlastnosti OpenGL specifikace. V tomto sdružení vystupovaly známé firmy, jako jsou ATI, NVIDIA, Apple, IBM, Intel a mnoho dalších firem, které se podílely na vývoji grafického hardwaru. [2]

V dnešní době, již od roku 2006, je sdružení OpenGL ARB vedeno jako část sdružení The Khronos Group. Veškerá správa OpenGL standardu nyní spadá pod tuto společnost. [2]

Základem OpenGL je takzvaná vykreslovací pipeline (angl. rendering pipeline), která je složena z jednotlivých fází a která umožňuje transformaci vstupních dat na obrazovou informaci. V dnešním moderním OpenGL je možné několik fází této pipeline programovat pomocí takzvaných shaderů, což jsou programy pro grafickou kartu. Tyto shadery jsou psány ve speciálním jazyce OpenGL Shading Language (zkr. GLSL) , který s OpenGL úzce souvisí. Shadery se liší v použité syntaxi jazyka GLSL podle fáze, pro kterou jsou určeny. [2]



Obrázek 2.1: Zjednodušená vykreslovací pipeline

Podrobnější informace o vývoji, využití, multiplatformnosti a ostatních vlastnostech OpenGL je možné najít v teoretické části, v příloze této práce.

Kapitola 3

Objekty

V případě OpenGL je objekt chápán jako uspořádání určitého počtu ploch do požadovaného geometrického útvaru. Výjimku pak tvoří body a čáry, které nevytváří žádné plochy. Objekty jsou základní prvky grafické scény a mohou být jak dvourozměrné, tak trojrozměrné. I přesto, že je dvourozměrná grafika stále využívána, je v dnešní době věnována větší pozornost trojrozměrnému prostoru a tedy i trojrozměrným objektům. Objekty, které jsou popsány souborem vrcholů, tvořících jednotlivé hrany a plochy, jsou nazývány mesh objekty. OpenGL pracuje právě s tímto typem objektů. Ostatní objekty, které mohou být popsány různými způsoby, například parametricky, je nutné na tento typ převést.

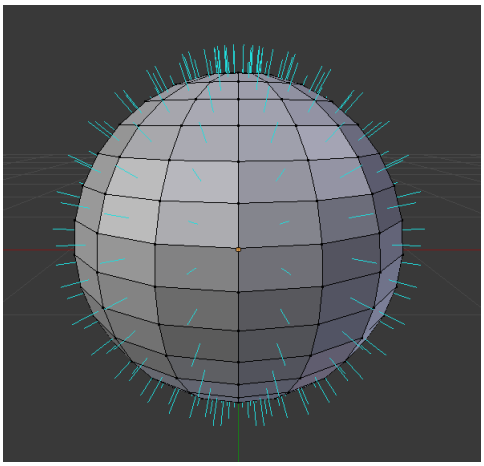
3.1 Vlastnosti objektů

Minimální vlastností, které musí mesh objekty nabývat, je vlastnost, obsahující informace o pozicích jednotlivých vrcholů, včetně informace o tom, které vrcholy společně formují jednotlivé plochy objektu. Vlastnosti se mohou lišit v tom, pro jakou úroveň objektu jsou definovány. Je tedy možno dělit vlastnosti na úrovni vrcholů, ploch nebo celého objektu. Mimo informace o pozicích vrcholů a plochách z nich tvořených je možné, aby pro objekt byly definovány různé vlastnosti, jako jsou normálové vektory, vhodné pro výpočet vlivu světla, barva, texturovací souřadnice, v případě že je na objekt mapována textura, nebo koeficienty pro různé výpočty. V OpenGL jsou objekty zpracovávány po vrcholech (per-vertex), z čehož plyne, že i přestože může být určitá vlastnost definována pro plochu, je tato vlastnost přidělena vrcholům, které ji tvoří (např. normálový vektor).

3.2 Vytváření objektů

Definování dat objektu přímo ve zdrojovém kódu aplikace je možný, avšak ne příliš vhodný způsob vytváření objektů, z důvodu časové náročnosti a následného znepřehlednění kódu. Proto je v praxi vhodnější využít externí grafické programy, specializované právě na tvorbu jednoduchých i složitých objektů (Blender, Maya, 3D Studio Max aj.). Modelovací nástroj Blender byl použit pro tvorbu objektů, použitých v ukázkových aplikacích této práce. Data objektů, vytvořených v těchto specializovaných programech, jsou exportována do souborů různých formátů

(.obj, .dae, .3ds, .ply atd.), které jsou poté v aplikaci přečteny a získaná data jsou vhodným způsobem uložena v paměti pro další práci.



Obrázek 3.1: Objekt se zobrazením normálových vektorů (Blender)

3.3 Zpracování objektů

Čtení dat z externích souborů, která představují objekt, vytvořený pomocí externího modelovacího programu, probíhá pomocí takzvaného parseru. Parser je program, který zná formát uložení dat a je schopný tato data získat. Složitost parseru se odvíjí právě od formátu souboru, ve kterém jsou data objektu uložena, neboť různé typy souboru představují různý formát uložení dat. Některé typy ukládají data v určitém formátu textové podoby, jiné spoléhají na uložení v podobě standardu XML. Pro parsování dat z různých typů souboru je možné využít externích knihoven, jejichž účelem je zpracování dat ze souboru jednoho nebo více formátů.

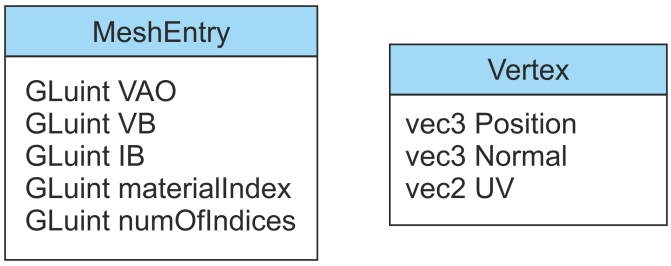
3.4 Implementace zpracování dat pomocí knihovny Assimp

Pro čtení dat objektů existuje množství knihoven (trimesh2, lib3ds, The OpenGL OBJ Loader aj.), avšak v této implementaci je použita open-source knihovna Assimp pro její schopnost zpracovat mnoho formátů [4]. Nevýhodou knihovny Assimp a podobných externích knihoven všeobecně je nutnost znát strukturu uložení, ve které jsou přečtená data nabízena programátorovi. Knihovna Assimp definuje několik desítek struktur, které mohou reprezentovat různé položky ve scéně (objekty, světla, kameru aj.).

Výsledkem zpracování souboru by měla být data objektu použitelná pro inicializaci příslušného array bufferu, který je možné doplnit index bufferem. Podrobnější popis jednotlivých typů bufferů je možné nalézt v [2]. Při zpracovávání souboru není nutné ukládat všechna data objektu, pouze ta, která jsou pro daný program nutná, od čehož se odvíjí podoba struktury, která sjednocuje vlastnosti pro jeden vrchol. Pro pozdější využití v dalších ukázkových příkladech jsou, mimo pozice vrcholů, uloženy i data texturovacích souřadnic a normálových vektorů.

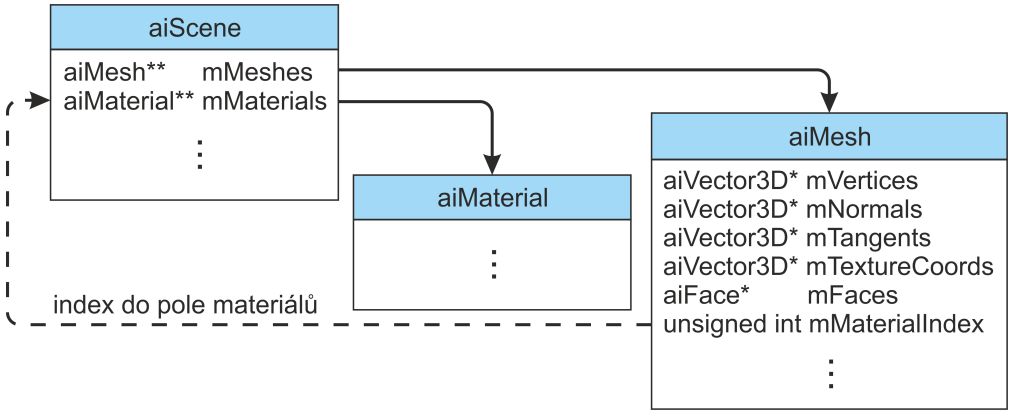
Problém, který může nastat u některých externích souborů, je ten, že mohou obsahovat data pro více objektů, jelikož je do těchto souborů exportována celá scéna. Je tedy nutné

rozhodnout, zda budou data jednotlivých objektů uložena ve vlastních bufferech nebo naopak bude vytvořen společný buffer pro uložení dat všech objektů dohromady. V této ukázkové implementaci je pro každý objekt vyhrazen vlastní buffer doplněný index bufferem, přičemž je vytvořena struktura, která obsahuje identifikátory těchto bufferů a další potřebná data pro objekt. K používání bufferu se v moderním OpenGL váže i povinnost použití takzvaného vertex array object (zkr. VAO), což je OpenGL objekt, který ukládá OpenGL stavy a popis rozložení dat v array bufferu [2]. Při vykreslování objektu je nutné znát počet vrcholů, které jej tvoří. V případě použití index bufferu je tento počet roven počtu indexů. Vytvoření struktury pro každý objekt má výhodu v možné změně stavů, například změně materiálu, mezi vykreslením jednotlivých objektů. Pro pozdější texturování objektu je v této struktuře obsažen i index materiálu, který indexuje do pole materiálů, vytvořeného knihovnou Assimp po zpracování souboru. Textury a texturování jsou předmětem následující kapitoly.



Obrázek 3.2: Struktura objektu a vrcholu

Základní struktura, kterou knihovna Assimp definuje, je struktura aiScene, která představuje celou scénu ze souboru. Pomocí této struktury je možné přistupovat k jednotlivým objektům scény, materiálům a dalším položkám [4]. Pro každou položku scény je definována struktura, která ji reprezentuje. Pro objekty je určena struktura aiMesh, ze které je možno získat požadovaná data objektu, pokud jsou pro daný objekt definována. Položka materiálu je reprezentována strukturou aiMaterial.



Obrázek 3.3: Propojení struktur

Průchodem všech struktur aiMesh, obsažených v poli struktury aiScene, jsou získána požadovaná data jednotlivých vrcholů objektu, kde pro každý vrchol je vytvořena struktura s příslušnými daty. Tato struktura je poté uložena do připraveného pole, které je později použito jako zdroj dat pro inicializaci array bufferu.

```
//saving material index for actual object
entries[index].materialIndex = mesh->mMaterialIndex;

std::vector<Vertex> vertices;
const aiVector3D zero3D(0.0,0.0,0.0);

//mesh is an object from loaded scene
for(GLuint i = 0; i < mesh->mNumVertices; i++)
{
    const aiVector3D *vPos = &(mesh->mVertices[i]);
    const aiVector3D *vNormal = &(mesh->mNormals[i]);
    const aiVector3D *vTexCoord = mesh->HasTextureCoords(0) ?
                                  &(mesh->mTextureCoords[0][i]) : &zero3D;

    Vertex v(
            glm::vec3(vPos->x,vPos->y,vPos->z),
            glm::vec3(vNormal->x,vNormal->y,vNormal->z),
            glm::vec2(vTexCoord->x,vTexCoord->y),
            );

    vertices.push_back(v);  //saving to array of object vertices
}
```

Pro získání dat, kterými je poté inicializován index buffer, je nutné procházet všechny plochy daného objektu a ukládat indexy vrcholů, které tvoří aktuální plochu. Struktura aiMesh obsahuje pole struktur aiFace, které reprezentuje všechny plochy objektu. Samotná struktura aiFace obsahuje požadované indexy vrcholů. Používání index bufferu pro vykreslování má obrovskou výhodu při práci s vrcholy, které nabývají stejných vlastností. Při použití index bufferu není nutné ukládat stejná data vícekrát, což by jinak způsobilo redundanci, neboť je dostačující uložit tato data pouze jednou a využít indexu pro opakované odkazování.

```
std::vector<GLuint> indices;

//mesh is an object from loaded scene
for(GLuint i = 0; i < mesh->mNumFaces; i++)
{
    const aiFace& face = mesh->mFaces[i];
    assert(face.mNumIndices == 3);
    indices.push_back(face.mIndices[0]);
    indices.push_back(face.mIndices[1]);
    indices.push_back(face.mIndices[2]);
}
```

Po každém získání dat jednoho objektu jsou tato data použita pro inicializaci bufferů daného objektu, s využitím OpenGL funkcí. Společně s inicializací bufferů jsou povoleny vstupní atributy pro vertex shader a je popsán formát uložení dat v array bufferu (velikost struktury jednoho vrcholu, offset ve struktuře pro dílčí vlastnosti). Popis uložení dat a všechny stavy,

včetně referencí na buffery, jsou uloženy do vertex array object, který je vytvořen a aktivován ještě před funkcemi, které by jej mohly ovlivnit. Všechny stavy jsou při příští aktivaci vertex array object automaticky nastaveny. Popis jednotlivých OpenGL funkcí je možno nalézt v [3].

```
glGenVertexArrays(1,&VAO); //Generating vertex array object ID
glBindVertexArray(VAO);

glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);

glGenBuffers(1,&VB); //Generating array buffer(vertex buffer) ID
glBindBuffer(GL_ARRAY_BUFFER,VB);
glBufferData(GL_ARRAY_BUFFER, sizeof(Vertex)*vertices.size(), &vertices[0],
             GL_STATIC_DRAW);

glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,sizeof(Vertex),0);
glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,sizeof(Vertex),(GLvoid*)12);
glVertexAttribPointer(2,2,GL_FLOAT,GL_FALSE,sizeof(Vertex),(GLvoid*)24);

glGenBuffers(1,&IB); //Generating index buffer ID
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,IB);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLuint)*indices.size(),
             &indices[0], GL_STATIC_DRAW);

glBindVertexArray(0);

numIndices = indices.size();
```

Jelikož výsledkem této ukázkové implementace je pouhé získání dat objektu bez jejich použití v shader programu, je ukázkový příklad spojen s implementací v následující kapitole, jejímž předmětem jsou textury. Výsledek načítání dat objektu a jejich použití, včetně mapování textur, je poté zobrazen na konci kapitoly o texturách.

Kapitola 4

Textury

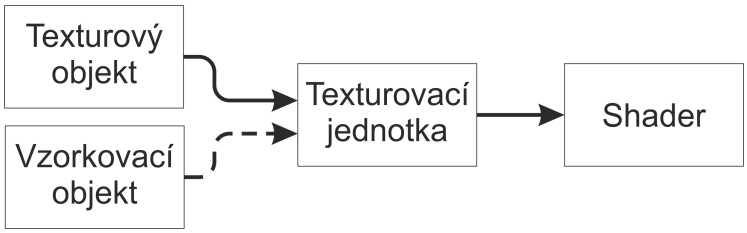
Obecně textury neodmyslitelně patří k počítačové grafice, ve které mají široké využití jak pro přiblížení vzhledu objektů realitě, tak pro různé techniky výpočtů, využívající textury jako zdroj nebo úložiště dat. Texturu je možné popsat jako několika-rozměrné pole, jehož dimenze závisí na typu textury. Jednotlivé texely, což jsou základní jednotky textury, mohou obsahovat různá data. Často tato data reprezentují barvu, přičemž pro takové textury OpenGL podporuje pouze barevné formáty RGB, RGBA a formáty z nich odvozené. [2]

4.1 Texturové objekty

V OpenGL jsou textury reprezentovány takzvanými texturovými objekty [2], které se skládají ze tří částí. Hlavní části odpovídá paměťové místo pro uložení dat textury. Další dvě části texturového objektu slouží pro uložení parametrů, které se dělí na parametry týkající se textury a na vzorkovací parametry použité při čtení texturových dat. Příkladem parametrů pro texturu je specifikace swizzling parametrů, které mohou způsobit prohození určitých barevných kanálů textury nebo jejich nahrazení hodnotou 0 nebo 1. Vzorkovací parametry mohou obsahovat různá nastavení pro čtení dat textury, jako jsou například různé typy filtrování nebo takzvaný wrapping. Filtrování a wrapping jsou podrobněji popsány v teoretické příloze této práce. Vzorkovací parametry mohou být texturovému objektu poskytnuty pomocí speciálního OpenGL vzorkovacího objektu (angl. sampler object), který tyto parametry ukládá. Asociací vzorkovacího objektu a texturového objektu ke stejné texturovací jednotce se parametry vzorkovacího objektu stávají aktuálními. Výhoda použití vzorkovacího objektu tkví v použití jednoho objektu pro více texturových objektů. [2]

4.2 Texturovací jednotky

Texturovací jednotky jsou komponenty grafické karty, které provádí proces vzorkování dat z textury a je možné je považovat za spojení mezi texturovým objektem, případně vzorkovacím objektem, a programem pro grafickou kartu, tedy shaderem. Pro každou grafickou kartu existuje určitý počet texturovacích jednotek, které je možné v aplikaci využívat. Moderní grafické karty obsahují průměrně kolem jedné stovky texturovacích jednotek.



Obrázek 4.1: Propojení jednotlivých částí v procesu texturování

4.3 Texturovací souřadnice

Texturovací souřadnice, známé také jako UV souřadnice, slouží k určení místa v textuře, ze kterého má být získán vzorek. Definují tedy určitou transformaci textury pro její mapování na danou plochu objektu. Aby bylo možné rozlišit texturovací souřadnice od prostorových souřadnic (x, y, z, w) a také od označení barevných složek (r, g, b, a), jsou v jazyce GLSL zpravidla označovány písmeny s, t, p, q. V závislosti na typu textury, která má být použita při vzorkování, se odvíjí počet komponent vektoru reprezentující texturovací souřadnice (2D textura – (s, t), 3D textura – (s, t, p) atd.). [2]

4.4 Vzorkování

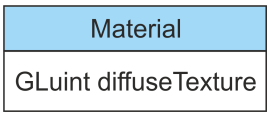
Pro získávání dat z textury, tedy vzorkování, definuje jazyk GLSL speciální built-in funkci *texture* [3]. Aby bylo možné tuto funkci použít, je nutné, aby byl texturový objekt asociován s texturovací jednotkou, jejíž číslo je poté nastaveno do speciální uniformní proměnné, která odpovídá určitému datovému typu ze skupiny datových typů *sampler*. Konkrétní typ proměnné je odvozen od typu textury, asociované s texturovací jednotkou (*sampler1D, sampler2D, samplerCube* aj.). Základní typy textur jsou popsány v teoretické příloze. Proměnná typu *sampler* společně s texturovacími souřadnicemi tvoří argumenty pro funkci *texture*, která je pro různé druhy *sampler* proměnných přetížena. [2]

V OpenGL je možné využívat funkcionality zvané Mipmapping, která se využívá pro zefektivnění procesu vzorkování a řešení určitých nežádoucích jevů. Popis a princip této funkcionality je možné rovněž najít v teoretické příloze.

4.5 Rozšíření objektů o mapování textur

Texturování objektů je základní vlastností OpenGL, kterou disponuje již od verze 1. Ukázková implementace v kapitole *Objekty* demonstrovala získání požadovaných dat pro jednotlivé vrcholy objektu, mezi kterýma byla i data texturovacích souřadnic. V návaznosti na tuto ukázku je popsán způsob implementace načítání textur a jejich využití. Pro implementaci je opět využito knihovny Assimp.

Jelikož je možné, aby byl materiál složen z více textur, je pro tento účel vytvořena struktura, sjednocující všechny textury daného materiálu. V této konkrétní implementaci obsahuje struktura materiálu pouze difúzní texturu, nicméně v pozdějších kapitolách této práce je využita i textura normálová a spekulární.



Obrázek 4.2: Struktura materiálu

Každý materiál, použitý na jednom nebo více objektech načtené scény, reprezentuje knihovna Assimp strukturou aiMaterial. Tato struktura poskytuje funkce pro zjištění informací o materiálu, jako jsou například počet textur daného typu, umístění textur, různé koeficienty, například odrazivost, a mnoho dalších [4]. Struktury aiMaterial, reprezentující všechny materiály použité ve scéně, jsou uloženy v poli, ve struktuře aiScene. Propojení struktur je znázorněno na obrázku *3.3* z předchozí kapitoly. Následující kód představuje způsob získání umístění difúzní textury.

```
const aiMaterial* mat = scene->mMaterials[i]; //getting material at index i

//Getting diffuse texture
if(mat->GetTextureCount(aiTextureType_DIFFUSE) > 0 )
{
    aiString path;

    if(mat->GetTexture(aiTextureType_DIFFUSE,0,&path,NULL,
                       NULL,NULL,NULL,NULL) == AI_SUCCESS)
    {
        std::string fullPath = std::string(path.C_Str());
        GLuint slashIndex = fullPath.find_last_of("\\");
        fullPath = "..\\Others\\Textures\\" +
                   fullPath.substr(slashIndex+1,fullPath.length());

        //Loading texture and saving texture ID to material structure
        materials[i].diffuseTexture = LoadTexture(fullPath);
    }
}
```

Samotné textury jsou reprezentovány jako externí soubory, pro jejichž načítání neposkytuje OpenGL žádnou funkcionalitu. Z tohoto důvodu je potřeba použít externí knihovnu pro tento účel. Takových knihoven je veliké množství, lišící se převážně v možnostech, které nabízí. Jedna z nejjednodušších knihoven pro načítání textur je například knihovna SOIL (Simple OpenGL Image Library), avšak v této ukázkové implementaci je využita knihovna OpenCV [5], která naopak nabízí široké a užitečné možnosti práce s texturou (např. zobrazení textury). Od OpenGL verze 4.2 jsou povinnou součástí implementace funkce pro alokaci neměnného paměťového místa (angl. immutable storage) texturového objektu. Tyto funkce se doporučuje použít, neboť zabraňují vzniku neúplného texturového objektu nebo nepovolených stavů (např. různé formáty pro úrovně mipmapy) [2]. Jedná se o funkce *glTexStorage*, které alokují místo pro celý texturový objekt najednou. Při použití mipmappingu je potřeba zadat počet úrovní pro vytvoření dostatečného místa. V následujícím kódu jsou načtena data textury a uložena do paměťového místa texturového objektu společně s dalšími automaticky vygenerovanými texturami pro mipmap úrovně. Nakonec jsou nastaveny vhodné vzorkovací parametry.

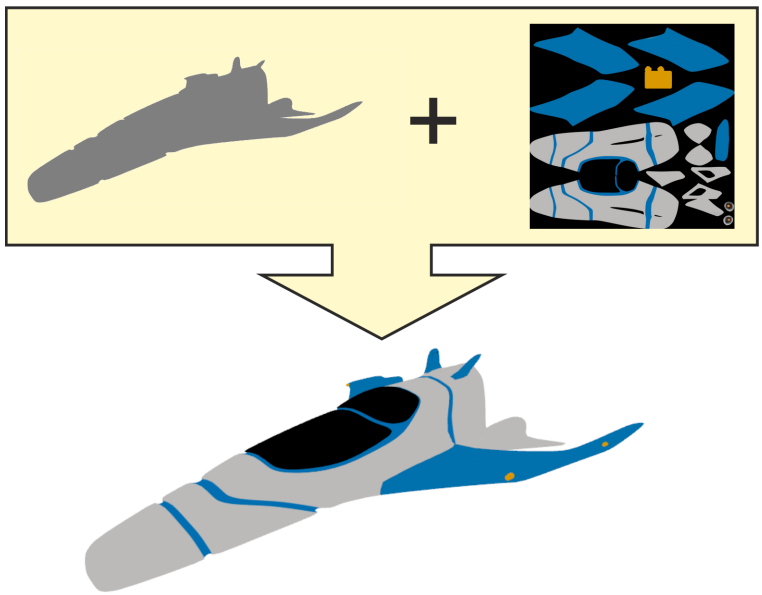
```
GLuint texture;

cv::Mat image = cv::imread(texturePath.c_str(), CV_LOAD_IMAGE_UNCHANGED );
cv::flip(image,image,0); // Y flipping

glGenTextures(1, &texture); //generating texture object ID
glBindTexture(GL_TEXTURE_2D, texture);

//Calculating number of mipmap levels and preparing formats
GLint levels = floor(log(MAX(image.cols,image.rows)) / log(2)) + 1;
GLuint format = (image.channels() == 3) ? GL_BGR : GL_BGRA;
GLuint internalFormat = (image.channels() == 3) ? GL_RGB8 : GL_RGBA8;

//Creating immutable storage and filling it with base texture data
glTexStorage2D(GL_TEXTURE_2D, levels, internalFormat, image.cols, image.rows);
glTexSubImage2D(GL_TEXTURE_2D,0,0,0,image.cols,image.rows,format,
                GL_UNSIGNED_BYTE, image.data);

glGenerateMipmap(GL_TEXTURE_2D);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
                GL_LINEAR_MIPMAP_LINEAR);

image.release();
```

Před samotným vykreslením objektu je nutné, aby byl nastaven jeho materiál. Texturový objekt, obsahující texturu daného materiálu, je asociován s texturovací jednotkou, jejíž číslo může nabývat hodnoty z intervalu $< 0, MAX\_TEXTURE\_UNIT)$, kde MAX_TEXTURE_UNIT je maximální počet texturovacích jednotek konkrétní grafické karty. Je také nutné nastavit požadované OpenGL stavy, což se děje aktivací příslušného vertex array object.

```
glBindVertexArray(entries[i].VAO);  //VAO of object at index i
GLuint matIndex = entries[i].materialIndex;

if(matIndex < materials.size())
{
    if(materials[matIndex].diffuseTexture != INVALID_MATERIAL)
    {
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, materials[matIndex].diffuseTexture);
    }
}

glDrawElements(GL_TRIANGLES, entries[i].numIndices, GL_UNSIGNED_INT, 0);
glBindVertexArray(0);
```

Pro práci s texturami je nejčastěji využíván fragment shader, nicméně textury mohou být použity i v ostatních programovatelných fázích vykreslovací pipeline. Na základě texturovacích souřadnic a *sampler* proměnné je získán vzorek z textury, jehož výsledná hodnota je ovlivněna všemi vzorkovacími i texturovými parametry. Menší výhodou od OpenGL verze 4.2 je možnost, pomocí layout specifikátoru a parametru binding, specifikovat defaultní asociaci s texturovací jednotkou pro *sampler* proměnnou. Následující kód představuje proces texturování ve fragment shaderu.

```
#version 430
in vec2 ex_UV;  //texture coordinates
out vec4 gl_FragColor;
layout(binding = 0)uniform sampler2D colorTexture; //sampler for texture

void main(void)
{
    gl_FragColor = texture(colorTexture, ex_UV);
}
```



Obrázek 4.3: Výsledný otexturovaný objekt

Při pohledu na výsledný obraz působí model velmi nerealisticky, což je způsobeno převážně absencí osvětlení. Absence osvětlení způsobuje konstantní intenzitu barvy po celém objektu. Z tohoto důvodu je předmětem následující kapitoly způsob simulace osvětlení.

Funkční ukázkový příklad načítání objektů a texturování je možno nalézt v příloze této práce. Konkrétní umístění je /Examples/01 - 02 Objects and textures .

Pro tuto kapitolu je vytvořen i další ukázkový příklad využívající algoritmus Volume ray casting pro zobrazování dat 3D textury. Tento příklad je vytvořen pouze jako doplňující ukázka a není v této práci popsán. Umístění příkladu je v příloze, ve složce /Examples/03. Volume rendering .

Kapitola 5

Osvětlení

Výpočet vlivu světla na objekty je, po použití textur, další z hlavních částí počítačové grafiky, která zvyšuje míru realističnosti výsledné scény. Existují dva druhy výpočtu vlivu světla, které se liší převážně v časové náročnosti.

Časové náročnější metody výpočtů jsou většinou založeny na fyzikálních zákonech šíření světla v prostoru. Příklady takových algoritmů jsou Ray Tracing, Path Tracing, Radiosity aj. U globálních osvětlovacích metod, jak jsou také tyto algoritmy nazývány, je hlavní příčinou vysoké časové náročnosti výpočet, do kterého jsou zahrnuty všechny okolní objekty, které mohou intenzitu výsledného světla ovlivnit. Z důvodu časové náročnosti není možné tyto metody využívat v real-time aplikacích. Existují však případy, kdy je možné se setkat s real-time aplikacemi, které tyto metody využívají, avšak nejedná se o plnohodnotné implementace těchto metod.

Druhý způsob výpočtu definují takzvané empirické osvětlovací modely. Tyto modely jsou využívány v real-time aplikacích z důvodu vysoké rychlosti výpočtu, která je způsobena určitým odstoupením od fyzikálních zákonů šíření světla. I přesto, že není dosaženo takové míry realističnosti jako u globálních osvětlovacích metod, je výsledek použití empirických osvětlovacích modelů stále přijatelný. Příklady empirických osvětlovacích modelů jsou Lambertův osvětlovací model nebo Phongův osvětlovací model, který je předmětem i ukázkové implementace.

5.1 Empirické osvětlovací modely

Výpočet osvětlení je, nejen v empirických osvětlovacích modelech, závislý na vektorech, známých jako normálové vektory, a proto je nutné, aby byly tyto vektory pro objekt definovány. Normálové vektory je možné vypočítat pomocí operace vektorového součinu, avšak v dnešní době většina modelovacích nástrojů podporuje automatické generování těchto vektorů. Mimo vektorový součin se při výpočtu osvětlení využívá několik dalších vektorových operací, jejichž popis je společně s popisem normálových vektorů obsažen v teoretické příloze.

Samotné empirické osvětlovací modely se rozlišují tím, které typy odrazů světla jsou zahrnuty v jejich výpočtu a také tím, jak jsou tyto odrazy vypočítávány. Každý typ odrazu je možné považovat na složku výsledného odrazu, který udává výslednou intenzitu světla v daném bodě. Popis jednotlivých složek je obsažen v teoretické příloze.

Lambertův osvětlovací model

Lambertův osvětlovací model je empirický model, který do výpočtu zahrnuje pouze difúzní odraz světla, avšak je možné se setkat i s rozšířením, kdy je zahrnut i odraz ambientní. Jedná se o nejjednodušší osvětlovací model, kde výsledná intenzita bodu závisí na tom, pod jakým úhlem dopadá světelný paprsek vůči normálovému vektoru daného bodu. Výsledný vzorec pro výpočet difúzního odrazu vychází z Lambertova zákonu.

$$I = \sum_{k=1}^{n} I_{L_k} k_d (\vec{L_i} \cdot \vec{N}) \tag{5.1}$$

n - *počet světel*
I_L - *barevné složení dopadajícího paprsku*
k_d - *koeficient difúzního odrazu*
$\vec{L_i}$ - *opačný vektor směru dopadajícího paprsku*
$\vec{N}$ - *normálový vektor*

Vzorec pro výpočet intenzity podle Lamberta [7]

Phongův osvětlovací model

Phongův osvětlovací model byl představen v roce 1973 Bui Tuong Phongem jako výsledek jeho disertační práce. Tento model vychází z Lambertova osvětlovacího modelu a rozšiřuje výpočet o ambientní a zrcadlový odraz světla. Výsledná intenzita pak odpovídá součtu množství světla odraženého všemi třemi typy odrazu. Tento model je často využíván v real-time aplikacích pro svou jednoduchost, rychlost a přijatelné výsledky.

$$I = I_A k_a + \sum_{l=1}^{n} I_L (k_d (\vec{L_i} \cdot \vec{N}) + k_s (\vec{R_i} \cdot \vec{V})^h) \tag{5.2}$$

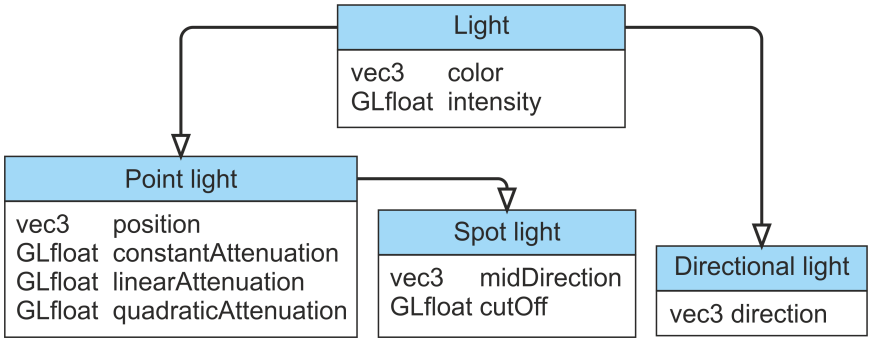
n - *počet světel*
I_A - *množství okolního světla*
I_L - *barevné složení dopadajícího paprsku*
k_a - *koeficient ambientního odrazu*
k_d - *koeficient difúzního odrazu*
k_s - *koeficient zrcadlového odrazu*
$\vec{L_i}$ - *opačný vektor směru dopadajícího paprsku*
$\vec{N}$ - *normálový vektor*
$\vec{R_i}$ - *vektor směru odraženého paprsku*
$\vec{V}$ - *vektor směru od místa odrazu k pozorovateli*
h -*Phongův exponent*

Vzorec pro výpočet intenzity podle Phonga [7]

5.2 Implementace osvětlení grafické scény

Tato ukázková implementace představuje využití Phongova osvětlovacího modelu pro tři různé typy světel. Jedná se o směrové světlo, bodové světlo a světlo typu reflektor, jejichž podrobnější popis je obsažen v teoretické příloze.

Podstatná část celkového výpočtu osvětlení je obsažena ve fragment shaderu příslušného shader programu, což odpovídá takzvanému Phongovému stínování, využívající interpolaci normálových vektorů. Nicméně je na straně klienta, tedy aplikace, potřeba připravit data, která odpovídají vlastnostem světel. Pro tyto účely jsou vytvořeny struktury, které obsahují položky různých vlastností na základě typu světla.



Obrázek 5.1: Struktury světel a dědičnost mezi nimi

V této implementaci je pro každý typ světla vytvořena jedna instance příslušné struktury a poté je naplněna vhodnými daty. Následovně jsou tato data poskytnuta fragment shaderu v podobě uniformních proměnných, jejichž datové typy odpovídají strukturám různých typů světel, které jsou ve fragment shaderu opětovně definovány. Mimo to jsou fragment shaderu poskytnuty informace o pozici pozorovatele, tedy pomyslné kamery, vlastnostech materiálu pro výpočet zrcadlové složky a také informace o intenzitě okolního světla pro výpočet ambientní složky, která se k výsledné intenzitě fragmentu přičítá pouze jednou a není závislá na konkrétním světelném zdroji. Mezi vlastnosti materiálu pro výpočet zrcadlové složky patří hodnota odrazivosti z intervalu $< 0, 1 >$ a Phongův exponent, který určuje ostrost výsledného efektu zrcadlové složky.

Aby byl výpočet vlivu světla pro daný fragment korektní, je nutné jej provádět na stejné prostorové úrovni. Jelikož jsou určité vlastnosti světel definovány v globálním prostoru, je nutné, aby i pozice fragmentu a jeho normálový vektor byl v tomto prostoru také. Pomocí modelové matice jsou, ve vertex shaderu, normálový vektor a pozice vrcholu přeneseny do globálního prostoru, ve kterém jsou poté interpolovány pro jednotlivé fragmenty. Následující kód představuje tuto transformaci.

```
void main(void)
{
    //in_Normal and in_Position are input attributes
    //ex_Normal and ex_WorldPos are output attributes
    ex_Normal = (modelMatrix * vec4(in_Normal,0.0)).xyz;
    ex_WorldPos = (modelMatrix * vec4(in_Position,1.0)).xyz;
}
```

Jelikož není ambientní složka závislá na konkrétním světle, je pro každé světlo vypočítána pouze složka difúzní a zrcadlová. K tomuto účelu je vytvořen společný kód pro všechny typy světla, který implementuje výpočet těchto dvou složek. Výpočet zrcadlové složky je doplněn koeficientem ze spekulární textury pro zvýšení realističnosti. Spekulární textura ukládá hodnoty koeficientu odrazu pro jednotlivé texely, čehož je využíváno v případech, kdy určité části mapované barevné textury mají být méně lesklé než části ostatní. Odrazivý koeficient pro výpočet difúzní složky je dán barvou fragmentu.

```
vec3 normal = normalize(ex_Normal);       //fragment normal
vec3 dir = normalize(direction);          //light ray direction

//Diffuse
float diffuseFactor = max(dot(normal, -dir),0.0);
vec3 diffuse = light.color * light.intensity * diffuseFactor;

//Specular
vec3 toEye = normalize(cameraPosition - ex_WorldPos);
vec3 reflected = normalize(reflect(dir, normal));
float specularFactor = max(dot(reflected, toEye),0.0);
vec3 specular = light.color * light.intensity *
                pow(specularFactor, specularFocus);

//specularCoeff is material shininess
vec3 color = min(colorTexel.rgb * diffuse + specularCoeff *
             specularTexel * specular, vec3(1.0));
```

Pro směrové světlo je tento společný výpočet ekvivalentní s celkovým výpočtem difúzní a zrcadlové složky, jelikož je pro všechny fragmenty směr svitu stejný a nedochází k pozdější modifikaci těchto dvou složek jako u ostatních typů světla.

Bodové světlo poskytuje pro každý fragment individuální směr příchozího paprsku, který je určen směrem od pozice světla k pozici fragmentu v globálním prostoru. Při použití bodového světla dochází k útlumu v závislosti na vzdálenosti pozice fragmentu od pozice světla. Výsledný útlum ovlivňuje hodnotu difúzní a zrcadlové složky ze společného výpočtu. Výpočet útlumu závisí na útlumových parametrech světla, které udávají způsob, jakým se útlum zvyšuje s rostoucí vzdáleností.

$$A = \frac{1}{k_c + k_l \cdot d + k_q \cdot d^2} \tag{5.3}$$

A - *výsledný útlum*
d - *vzdálenost*
k_c - *konstantní koeficient útlumu*
k_l - *lineární koeficient útlumu*
k_q - *kvadratický koeficient útlumu*

Vzorec pro výpočet útlumu [7]

Vzorec útlumu obsahuje lineární a kvadratický koeficient, které mohou být libovolně měněny podle vlastních požadavků. Konstantní koeficient slouží pro eliminaci působení vyšší intenzity světla, než samotné světlo vyzařuje, při velmi nízké vzdálenosti fragmentu od světla. Získání individuálního směru pro fragment a aplikaci útlumu znázorňuje následující kód.

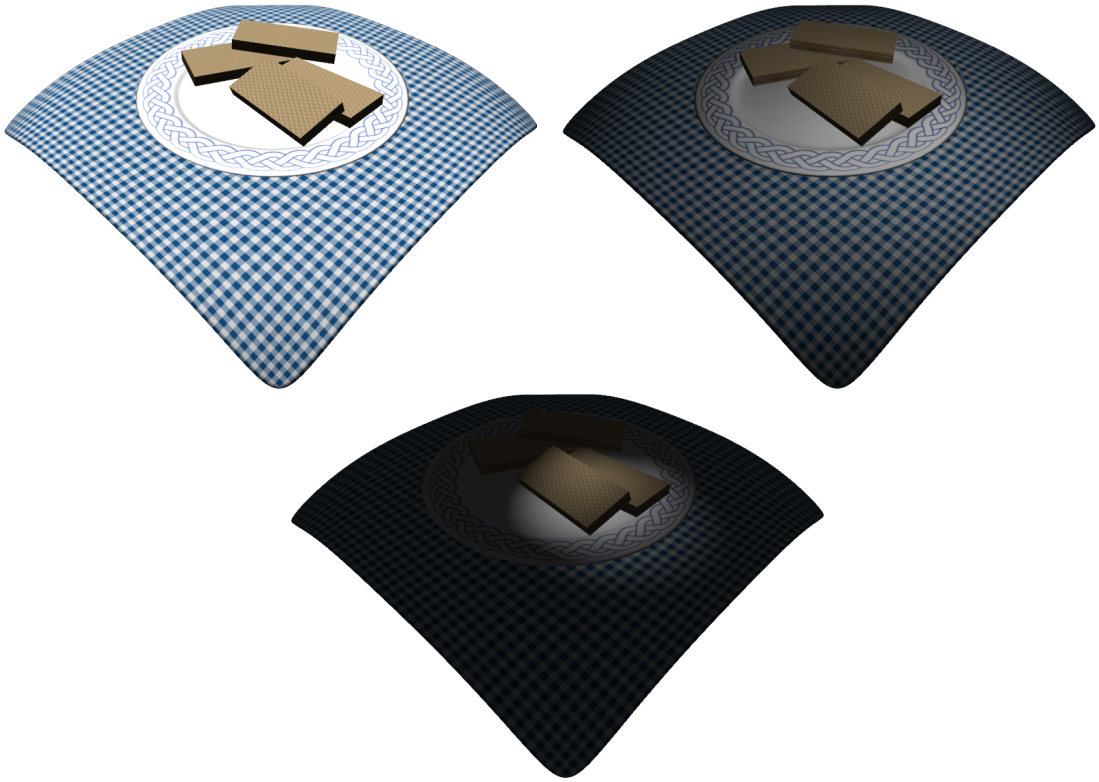
```
vec3 direction = ex_WorldPos - pointLight.position; //individual direction

//General calculation for diffuse and specular part
vec4 color = CalcBaseLight(pointLight.base, direction, colorTexel,
                           specularTexel);

float dist = length(direction);
float attenuation = pointLight.constantAttenuation +
                    pointLight.linearAttenuation * dist +
                    pointLight.quadraticAttenuation * pow(dist,2);

color /= attenuation;
```

Implementace reflektoru je velmi podobná bodovému světlu, nicméně výpočet difúzní a zrcadlové složky je proveden pouze pro omezený počet směrů. Provedení výpočtu těchto složek závisí na tom, zda je kosinus úhlu, který je svírán paprskem a středovým paprskem, v povoleném rozmezí. U reflektoru dochází k jevu, kdy intenzita slábne nejen se zvyšující se vzdáleností, ale také i se zvyšujícím se odklonem od středu světelného kužele. Pro tento účel je provedeno mapování intervalu $< \cos(max\_angle), 1 >$ do intervalu $< 0, 1 >$, kdy při nulovém odklonu je intenzita maximální a při odklonu, daným maximálním úhlem, je intenzita nulová. Následující kód představuje implementaci reflektoru.

```
vec4 color = vec4(0.0,0.0,0.0,1.0);
vec3 direction = ex_WorldPos - spotLight.position;  //individual direction
vec3 normMidDir = normalize(spotLight.midDirection);
vec3 normDir = normalize(direction);
float cosAngle = dot(normMidDir, normDir);

if(cosAngle >= spotLight.cutOff)      //if acceptable value
{
    //General calculation for diffuse and specular part
    color = CalcBaseLight(spotLight.base, direction, colorTexel,
                          specularTexel);
    //Mapping to [0,1] interval and its application
    color *= 1.0 - (1.0 - cosAngle) * (1.0 / (1.0 - spotLight.cutOff));

    float dist = length(direction);
    float attenuation = spotLight.constantAttenuation +
                        spotLight.linearAttenuation * dist +
                        spotLight.quadraticAttenuation * pow(dist,2);

    color /= attenuation;
}
```

K hodnotě součtu difúzní a zrcadlové složky, která určuje barvu fragmentu, je přičtena hodnota ambientní složky v podobě určitého množství bílého světla, které eliminuje velmi tmavá místa. Ambientní složka je, stejně jako složka difúzní, ovlivněna hodnotou texelu z barevné textury. Následující kód demonstruje konečný výpočet barvy fragmentu ve fragment shaderu.

```
void main(void)
{
    vec4 colorTexel = texture(colorTexture, ex_UV);
    float specularTexel = texture(specularTexture, ex_UV).r;

    //Calculating diffuse and specular part for chosen type of light
    vec4 DiffAndSpec = CalcLight(colorTexel, specularTexel);
    gl_FragColor = ambientIntensity * colorTexel + DiffAndSpec;
}
```



Obrázek 5.2: Výsledné osvětlení různých typů světla

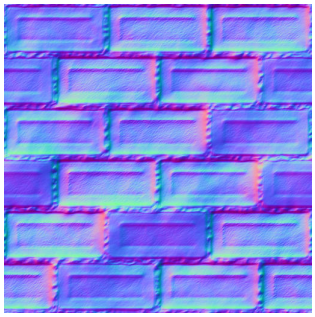
Výsledný obrázek znázorňuje osvětlení pomocí všech tří typů implementovaných světel. Výsledný osvětlený objekt vypadá realističtěji, nicméně na první pohled působí hladkým povrchem, což odporuje mapované textuře, reprezentující nerovný povrch. Způsob řešení je zvýšení počtu vrcholů objektu pro vytvoření geometrických nerovností nebo také změna optických vlastností pomocí techniky Normal mapping, která je implementována v následující kapitole.

Spustitelný ukázkový příklad je umístěn v příloze této práce, ve složce /Examples/04. Lighting .

Kapitola 6

Normal mapping

Normal mapping je technika, která je využívána pro simulaci nerovností povrchu objektu. Jedná se pouze o optický jev, který je způsoben určitou změnou normálového vektoru, což při výpočtu osvětlení ovlivní výslednou intenzitu a způsobí dojem nerovnosti. K technice Normal mapping se váže použití takzvané normálové textury, kde jednotlivé texely reprezentují normálový vektor s mírnou modifikací, v podobě barevné hodnoty. Pro každý fragment je z této textury získána příslušná hodnota, která je poté modifikována a použita jako normálový vektor při výpočtu osvětlení, kde nahrazuje původní interpolovaný normálový vektor.



Obrázek 6.1: Normálová textura

6.1 Tangent space

Tangent space, někdy také znám jako texture space, je trojrozměrný prostor, ve kterém jsou definovány normálové vektory v normálové textuře. Zpravidla je osa X souřadného systému prostoru reprezentovaná osou U textury a osa Y osou V. Třetí osou souřadného systému je kolmý vektor na rovinu XY.

Pro korektní výpočet osvětlení s normálovým vektorem, získaným z textury, je nutné provést transformaci z tangent space do lokálního prostoru objektu a následně do globálního prostoru, ve kterém jsou definovány ostatní parametry pro výpočet osvětlení. Nicméně je možné se setkat i s opačným postupem, kdy jsou parametry v globálním prostoru transformovány do tangent space, ve kterém poté probíhá výpočet. Transformace mezi tangent space a lokálním prostorem objektu je prováděna takzvanou Tangent-Bitangent-Normal maticí (zkr. TBN matice).

TBN matice je tvořena třemi jednotkovými vektory, které představují bázové vektory tangent space v lokálním prostoru. V matici TBN, pro transformaci do tangent space, jsou bázové vektory uloženy v řádcích. Při opačné transformaci je nutné takovou matici invertovat. Výhodou matice, která obsahuje navzájem ortogonální vektory, je ta, že její inverzní matice je ekvivalentní s maticí transponovanou.

$$\begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix} \tag{6.1}$$

T_x, T_y, T_z - *komponenty vektoru tangent*
B_x, B_y, B_z - *komponenty vektoru bitangent*
N_x, N_y, N_z - *komponenty normálového vektoru*

Tangent-Bitangent-Normal matice [8]
(Pro transformaci z tangent space do lokálního prostoru)

6.2 Implementace použití techniky Normal mapping

Následující ukázková implementace demonstruje získání a modifikaci normálového vektoru z normálové textury, který je poté možno použít pro výpočet osvětlení, které bylo předmětem implementace předchozí kapitoly.

Základem použití této techniky je získání požadovaných tří bázových vektorů tangent space. Aby však nedocházelo k ukládání redundantních dat, jsou dostačující pouze dva tyto vektory, z nichž může být poté získán třetí vektor pomocí vektorového součinu. Z tohoto důvodu je pro každý vrchol definován jen normálový vektor a vektor tangent, který může být, stejně jako normálový vektor, hodnotou aritmetického průměru tangent vektorů ploch, které daný vrchol sdílejí.

Získání vektoru tangent je možné několika způsoby. První možnost spočívá ve vytvoření funkce, která implementuje výpočet pro vektory tangent a bitangent na základě vzorce, uvedeného v teoretické příloze. Z tohoto výpočtu je poté ponechán pouze vektor tangent. Druhou možností je využití externí knihovny, která již implementaci výpočtu vektoru tangent obsahuje. V této ukázkové implementaci je opět využita knihovna Assimp, jejíž funkce, sloužící pro načtení souboru s daty objektu, přijímá různé příznaky, které specifikují dodatečné operace pro načtený objekt. Následující kód představuje použití této funkce s příznakem pro výpočet vektoru tangent. Vypočítaný vektor tangent je poté uložen ve struktuře aiMesh společně s ostatními daty objektu.

```
Assimp::Importer imp;

const aiScene* scene = imp.ReadFile(fileName, aiProcess_CalcTangentSpace);
```

Společně s ostatními daty pro vrchol je i vektor tangent poskytnut vertex shaderu, ve kterém je, stejně jako normálový vektor, transformován do globálního prostoru. Transformací těchto bázových vektoru do globálního prostoru je zajištěno, že výsledná TBN matice, která z nich bude vytvořena, bude transformovat z tangent space přímo do globálního prostoru. Část hlavní funkce vertex shaderu vypadá následovně.

```
void main(void)
{
    //in_Normal and in_Tangent are input attributes
    //ex_Normal and ex_Tangent are output attributes
    ex_Normal = (modelMatrix * vec4(in_Normal,0.0)).xyz;
    ex_Tangent = (modelMatrix * vec4(in_Tangent, 0.0)).xyz;
}
```

Interpolovaný normálový vektor a vektor tangent tvoří, ve fragment shaderu, základ pro modifikaci normálového vektoru, získaného z normálové textury. Mezi těmito dvěma bázovými vektory je proveden Gramův-Schmidtův ortogonalizační proces pro zajištění vzájemné ortogonality. Základní princip ortogonalizačního procesu je popsán v teoretické příloze. Všechny tří bázové vektory tangent space, kde vektor bitangent je získán vektorovým součinem, tvoří výslednou TBN matici. Datový typ *mat3* ukládá vektory, předané konstruktoru, do jednotlivých sloupců matice, což vytvoří požadovanou matici.

Poslední úpravou, před samotným použitím TBN matice, je úprava vzorku z normálové textury. Normálová textura ukládá hodnoty normálových vektorů v podobě barvy, jejíž hodnoty jsou pouze kladné, což není pro normálové vektory typické, jelikož mohou nabývat i záporných hodnot pro jednotlivé komponenty. Proto je hodnota z normálové textury převedena z intervalu $< 0, 1 >$ na interval $< -1, 1 >$. Následující kód představuje získání vzorku z normálové textury a jeho následnou modifikaci pro použití ve výpočtu osvětlení.

```
vec3 normalTexel = texture(normalTexture, ex_UV).xyz;

vec3 normal = normalize(ex_Normal); //input normal vector

vec3 tangent = normalize(ex_Tangent);    //input tangent vector
tangent = tangent - dot(normal, tangent) * normal;  //Gram-Schmidt process
tangent = normalize(tangent);

vec3 bitangent = cross(tangent, normal);
bitangent = normalize(bitangent);

mat3 TBN = mat3(tangent, bitangent, normal);

//mapping to [-1,1] and transformation from tangent space to global space
vec3 modifiedNormal = TBN * ((normalTexel - 0.5) * 2.0);
modifiedNormal = normalize(modifiedNormal);
```

Obrázek 6.2: Výsledná grafická scéna před a po použití techniky Normal mapping

Výsledný obrázek ukazuje scénu před a po použití techniky Normal mapping. Na pravém obrázku lze vidět, že povrch objektu působí dojmem mnoha nerovností, nicméně topologie objektu je stále stejná.

Spustitelná ukázková aplikace je umístěna v příloze této práce, ve složce /Examples/05. Normal mapping .

Kapitola 7

Stíny

Stíny jsou neodmyslitelnou součástí všech objektů, které jsou jakýmkoliv způsobem osvětleny. V real-time aplikacích, které využívají empirické osvětlovací modely, však nedochází k vytváření stínů, neboť je výsledná intenzita osvětlení každé části objektu vypočítávána bez ohledu na ostatní části nebo okolní objekty. Řešením tohoto problému jsou techniky, jejichž účelem je dodatečně určit, které části výsledné scény jsou ve stínu, a které naopak ne. Stejně jako u výpočtu osvětlení, využívající Phongova stínování, pracují jednotlivé techniky pro výpočet stínů na úrovni fragmentů. Nicméně i tyto techniky jsou používány pro svou rychlost výpočtu na úkor výsledného realistického vzhledu i přesto, že existuje mnoho jejich vylepšení.

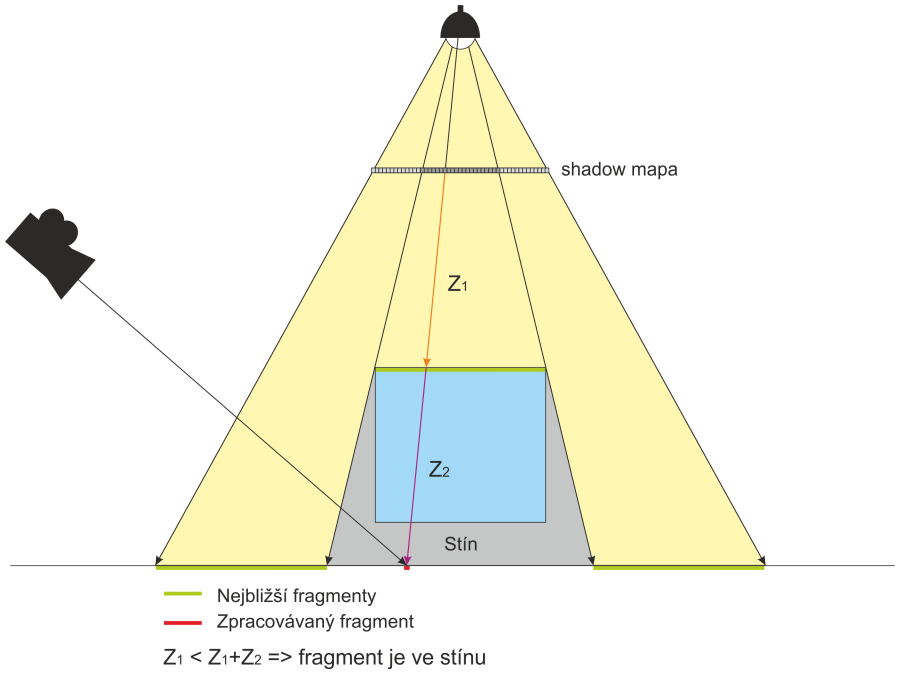
Základními a často používanými technikami pro výpočet stínu jsou techniky Shadow mapping a Stencil shadow volume, které jsou také předmětem ukázkových implementací této kapitoly.

7.1 Shadow mapping

Shadow mapping je často používaná technika, která pro výpočet využívá takzvanou shadow mapu. Shadow mapa je speciální textura, kde každý texel obsahuje pouze jednu hodnotu s plovoucí desetinnou čárkou. Princip techniky je založen na porovnávání hodnoty hloubky aktuálně zpracovávaného fragmentu s hodnotou hloubky uloženou v shadow mapě, kde obě tyto hodnoty jsou určené vzhledem k pozici světelného zdroje. Hodnoty, uložené v shadow mapě, reprezentují hodnoty hloubek fragmentů, které jsou v nejmenší vzdálenosti od světelného zdroje, což zajišťuje hloubkový test. Shadow mapping je multifázová technika, neboť je její použití prováděno ve dvou fázích. [9]

Účelem první fáze je získání hodnot hloubek nebližších fragmentů vzhledem ke světelnému zdroji. Získání těchto hodnot se provádí vykreslením dané scény z pozice světelného zdroje, při kterém dochází k uložení nejmenších hloubek do shadow mapy. Výsledná shadow mapa s požadovanými hodnotami je následovně využita v druhé fázi této techniky.

Druhá fáze reprezentuje výpočet osvětlení podle daného empirického osvětlovacího modelu, kde je tento výpočet rozšířen o porovnávání hodnot hloubek, jehož výsledek je poté zakomponován do výpočtu výsledné intenzity fragmentu. Pro zastíněný fragment je intenzita osvětlení snížena o určitou hodnotu, což ve výsledku vytváří dojem stínu.



Obrázek 7.1: Princip techniky Shadow mapping

7.1.1 Odlišnosti v použití

Použití techniky Shadow mapping se mírně liší na základě použitého typu světelného zdroje. Hlavní princip techniky je pro všechny typy stejný, nicméně pro každý typ je určena modifikace techniky tak, aby výsledné stíny odpovídaly vlastnostem daného typu světelného zdroje.

Pro směrové světlo se v první fázi techniky používá ortogonální neboli pravoúhlá projekce. Při ortogonální projekci jsou všechny paprsky, dopadající na průmětnu, vysílány stejným směrem, což také odpovídá směru paprsků, které vyzařuje směrové světlo. Pro shadow mapu je dostačující, aby byla reprezentována 2D texturou. Jelikož směrové světlo nenabývá konkrétní pozice, která je nutná pro vykreslení scény z pohledu světla, je pozice určena opačným vektorem určující směr svitu.

U světelného zdroje typu reflektor se používá naopak projekce perspektivní, která přesněji odpovídá vlastnostem vyzářených paprsků. Reflektor nabývá jak pozice, tak směru svitu, což jsou potřebné parametry pro určení pohledu do scény z pozice světelného zdroje. Pro shadow mapu je opět dostačující, aby byla reprezentována 2D texturou.

Pro bodové světlo nabývá technika Shadow mapping největších změn. Jelikož jsou světelné paprsky vyzařovány do všech směrů, je shadow mapa reprezentována cube map texturou pro pokrytí celého okolí světelného zdroje. Stejně jako u typu reflektor, je i zde použita perspektivní projekce. Pro pokrytí všech směrů jsou v první fázi techniky zapsány hodnoty do všech šesti textur, tvořících cube map texturu, kde každá textura odpovídá jednomu pohledu (vpravo, vlevo, dopředu, dozadu, nahoru, dolů).

7.1.2 Implementace techniky Shadow mapping

Následující ukázková implementace představuje hlavní části implementace techniky Shadow mapping pro bodové světlo, které bylo vybráno z důvodu mírně složitější implementace oproti ostatním typům světelného zdroje a také pro pozdější možnost porovnávání výsledků této techniky a techniky Stencil shadow volume, pro kterou je rovněž vybráno bodové světlo jako předmět implementace.

Pro první fázi této techniky je základním prvkem vytvoření texturového objektu, reprezentujícího shadow mapu, a jeho následné připojení k framebufferu. Framebuffer je OpenGL objekt, který obsahuje až čtyři různé buffery do různé účely. Podrobnější popis je možné nalézt v [2]. Důležitou vlastností framebufferu je možnost připojení textury jako jednu z možných komponent. Při použití této techniky je shadow mapa připojena jako hloubková komponenta (angl. depth component), což znamená, že budou do této textury ukládány výsledky hloubkového testování, které, po zpracování všech fragmentů, budou odpovídat hloubkám nejbližších fragmentů.

Pro zjednodušení pozdějšího porovnávání hodnot nabízí OpenGL hardwarovou podporu pro vzorkování z textury s následným porovnáním vzorku s referenční hodnotou [2]. Pro využívání této funkcionality je nutné pro texturový objekt povolit porovnávací mód a určit způsob porovnávání. Následující kód představuje vytvoření texturového objektu, pro něhož je alokováno neměnné paměťové místo, a jeho následné propojení s vytvořeným framebufferem, včetně vhodného nastavení.

```
glGenTextures(1, &shadowCubeMap);    //generating texture object ID
glBindTexture(GL_TEXTURE_CUBE_MAP, shadowCubeMap);

/* Setting sampling parameters */

//Parameters for comparing
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_COMPARE_FUNC, GL_LESS);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);

//Immutable storage for cube map texture
glTexStorage2D(GL_TEXTURE_CUBE_MAP,1,GL_DEPTH_COMPONENT32,mapSize,mapSize);

glGenFramebuffers(1, &framebuffer); //generating framebuffer object ID
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, shadowCubeMap,0);

glDrawBuffer(GL_NONE);   //no color buffer for drawing
glReadBuffer(GL_NONE);   //no color buffer for reading
```

Vytvořený framebuffer objekt je při první fázi techniky používán místo defaultního framebufferu, což způsobí, že jakýkoliv shader program bude používat tento framebuffer pro ukládání hodnot. Aktivace určitého framebufferu je prováděna pomocí funkce *glBindFramebuffer* [2][3].

Jelikož je shadow mapa reprezentována cube map texturou, je nutné, aby v první fázi byly zapsány požadované hodnoty hloubek do všech šesti textur cube map textury. Pro každou

texturu je specifikována pohledová matice, která určuje pohled od pozice světelného zdroje do daného směru z důvodu zachycení hloubek fragmentů objektů z celého svého okolí. Každá pohledová matice je matice pro perspektivní projekci s devadesáti stupňovým zorným úhlem a poměrem výšky a šířky definovaného frustra rovným hodnotě 1 z důvodu stejné šířky a výšky všech textur cube map textury.

Pro první fázi je scéna vykreslena šestkrát, pokaždé s jinou pohledovou maticí a pokaždé se zápisem do jiné textury. Mimo změny pohledové matice a textury jako hloubkové komponenty framebufferu je provedena změna velikosti viewportu z důvodu eliminace možné deformace obrazu, která vzniká při odlišném poměru stran viewportu a definovaného frustra. Změna viewportu je prováděna funkcí *glViewport* [2][3].

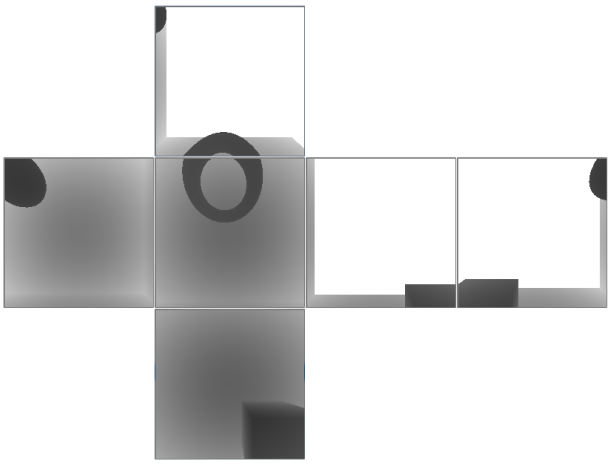
Po nastavení výše zmíněných věcí je provedeno vykreslení pomocí jednoduchého shader programu, jehož úkolem je pouhý zápis hloubky fragmentu. Implementace pro bodové světlo může tvořit výjimku v tom, že není zapisována hloubka určená přední a zadní ořezávací rovinou, jak je to automaticky prováděno v OpenGL, nýbrž je ukládána hloubka určená pozicí světelného zdroje a zadní ořezávací rovinou. Tento způsob je použit v této implementaci, k čemuž je nutná informace o pozici fragmentu v globálním prostoru. Následující kód představuje část hlavní funkce vertex shaderu pro získání této pozice.

```
void main()
{
    //in_Position is input attribute
    //ex_WorldPos is output attribute
    ex_WorldPos = (modelMatrix * vec4(in_Position, 1.0)).xyz;
}
```

Oproti fragment shaderům, obsažených v jiných shader programech, není výstupem fragment shaderu tohoto shader programu barva fragmentu, ale pouze jeho hloubka. Přístup k hloubce fragmentu je možný pomocí built-in proměnné *gl_FragDepth* [2][3]. V této implementaci je automaticky generovaná hloubka fragmentu přepsána na hodnotu vzdálenosti fragmentu od pozice světelného zdroje, která je následně převedena do intervalu $< 0, 1 >$ vydělením maximální hodnotou vzdálenosti, tedy hodnotou vzdálenosti zadní ořezávací roviny. Výhodou tohoto přepsání je poté jednodušší porovnávání vzdáleností fragmentů v druhé fázi techniky. Následující kód je část fragment shaderu demonstrující přepsání hodnoty hloubky fragmentu.

```
void main()
{
    float dist = distance(ex_WorldPos, lightPosition);
    gl_FragDepth = dist / farPlane; //division by far plane distance
}
```

Výsledek první fáze techniky Shadow mapping je shadow mapa, kde všech šest textur, tvořících cube map texturu, je naplněno požadovanými daty, které jsou v následující fázi použity jako referenční hodnoty při porovnávání.



Obrázek 7.2: Ukázka výsledné shadow mapy

Druhá fáze je rozšířením funkcionality fragment shaderu shader programu pro osvětlení, pro který je zpřístupněna shadow mapa z první fáze jako klasická textura pro vzorkování. Pro tuto fázi je již aktivován defaultní framebuffer a je možno vykreslit scénu z jakéhokoliv požadovaného pohledu.

Ve fragment shaderu probíhá porovnávání hloubky fragmentu, která je získána stejným způsobem, jakým byly získány hodnoty hloubky v první fázi. Tohle je výrazné ulehčení, neboť není nutné zjišťovat, který pohled zahrnuje daný fragment a poté na základě určitého pohledu vypočítat jeho hloubku. Směr od pozice světelného zdroje k fragmentu také určuje texturovací souřadnice pro vzorkování z cube map textury. Zde je využito, v úvodu implementace zmiňované, funkcionality pro porovnávání vzorkované hodnoty s referenční hodnotou. K tomuto účelu se využívá GLSL datových typů *shadowSampler* [2][3], kdy při jejich použití v texturovací funkci je nutné rozšířit texturovací souřadnice o referenční hodnotu. V tomto případě je referenční hodnota hloubka aktuálního fragmentu. Výsledek porovnávání nabývá hodnoty z intervalu $< 0, 1 >$, což je v tomhle případě i hodnota určující pokles intenzity difúzní a zrcadlové složky výsledné intenzity. Následující kód je část fragment shaderu pro určení koeficientu poklesu intenzity.

```
float shadowFactor = 0.0; //factor affecting final intensity
vec3 lightToFrag = ex_WorldPos - lightPosition; //cube map sampling vector
float actualZ = length(lightToFrag) / farPlane; //actual fragment depth

//shadowMap data type = samplerCubeShadow
shadowFactor = texture(shadowMap, vec4(lightToFrag, actualZ));
```

Problémem techniky Shadow mapping je to, že při jejím použití může vznikat mnoho nežádoucích jevů. Základní artefakty, jak se těmto jevům nazývá, a jejich možná řešení jsou popsány v teoretické příloze.

Výsledek implementace je zobrazen v pozdější sekci, kde je porovnáván s výsledkem techniky Stencil shadow volume.

7.2 Stencil shadow volume

Stencil shadow volume je další používaná technika pro vytváření stínů, jejíž princip je založen na objemových stínových tělesech [10]. Stínové těleso, které je vytvářeno pro každý objekt vrhající stín, reprezentuje prostor, který je daným objektem zastíněn. Cílem této techniky je určit, zda jsou jednotlivé fragmenty uvnitř stínového tělesa či ne. K tomuto účelu je využit stencil buffer, do kterého jsou na základě určitého algoritmu ukládány hodnoty, určující stav zastínění fragmentů. Hodnoty stencil bufferu jsou následně použity při vykreslování, kde slouží jako hodnoty pro takzvaný stencil test. Samotnou techniku je možné rozdělit do tří hlavních fází.

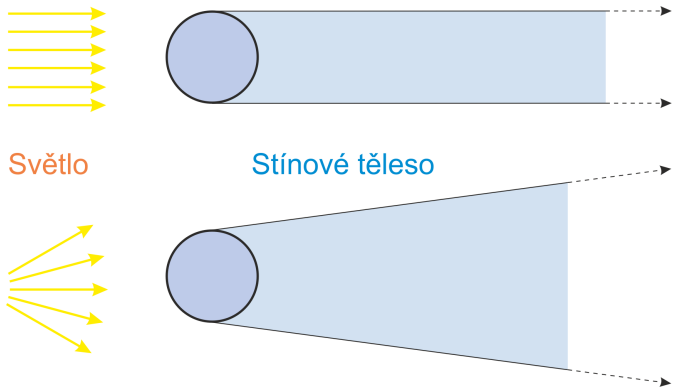
V první fázi dochází k uložení hodnot hloubek nejbližších fragmentů všech viditelných těles z libovolného pohledu do scény. Tyto hodnoty jsou ukládány do hloubkového bufferu a jsou poté použity pro určování hodnot stencil bufferu v další fázi této techniky.

Fáze druhá zajišťuje vytvoření stínových těles pro objekty, které vrhají stín. Fragmenty vytvořených těles jsou poté, pomocí hloubkového testu, porovnávány s hodnotami, uloženými v hloubkovém bufferu z první fáze. Na základě výsledků hloubkového testu je zvolena operace nad hodnotami stencil bufferu.

Třetí fáze představuje konečné vykreslení scény, které je ovlivněno hodnotami stencil bufferu. Tyto hodnoty určují, pro které fragmenty jsou, ve výpočtu výsledné intenzity, zahrnuty všechny složky nebo naopak pouze složka ambientní.

7.2.1 Odlišnosti v použití

Techniku Stencil shadow volume je možné používat pro všechny tři typy světelného zdroje, kde algoritmus pro oba typy světelného zdroje se nijak výrazně neliší. Mírná odlišnost nastává při vytváření stínových těles, kde je nutné tato tělesa vytvořit podle vlastností vyzářených paprsků světla. Určité hrany, které vytváří plášť stínového tělesa, jsou při použití směrového světla vzájemně rovnoběžné, neboť jsou vytvářeny podél stejného směru. Oproti tomu pro bodové světlo nebo reflektor se tyto hrany, se zvyšující se vzdáleností, rozbíhají.



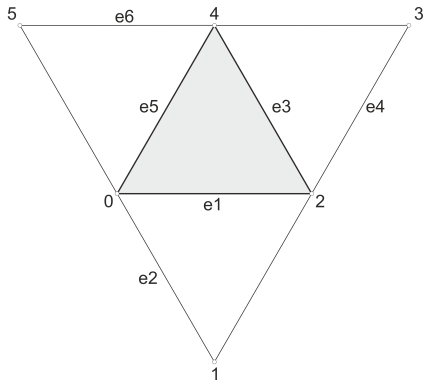
Obrázek 7.3: Různá stínová tělesa pro různé typy světla

Bez ohledu na typ světla se mohou implementace této techniky lišit podle zvoleného algoritmu pro určování hodnot stencil bufferu. Jedná se o algoritmy Z-pass a Z-fail, jejichž popis je obsažen v teoretické příloze.

7.2.2 Implementace techniky Stencil shadow volume

V této části je představena implementace techniky Stencil shadow volume pro bodové světlo. Tato implementace využívá algoritmu Z-fail, jehož použití je mírně složitější než použití algoritmu Z-pass.

Jedním z hlavních požadavků pro možnost použití této techniky je nutnost, aby každá plocha objektu měla informaci o sousedních plochách. Pro tento účel poskytuje OpenGL podporu pouze pro trojúhelníkové plochy, kde každá plocha je reprezentována šesti vrcholy, z čehož tři tvoří danou plochu a zbylé tři jsou třetí vrcholy sousedních ploch.



Obrázek 7.4: Indexace vrcholů a označení hran

Pro možnost specifikovat plochu jako soubor šesti vrcholů je při vykreslování použit speciální typ primitiva, a to *GL_TRIANGLES_ADJACENCY*. Vertex shader všechny tyto vrcholy zpracovává nezávisle na sobě, nicméně v geometry shaderu jsou již společně přístupné.

Jedna z možností pro získání vrcholů sousedních ploch je založena na sdílených hranách. Každý trojúhelník je složen ze tří hran, které jsou sdíleny jeho sousedními plochami. V principu je nutné získat pro každou hranu index dvou sousedních ploch. Při pozdějším zpracovávání konkrétní plochy objektu je pro všechny jeho hrany proveden dotaz na druhou plochu, která ji sdílí, a poté dotaz na vrchol druhé plochy, který není součástí sdílené hrany. Po zpracování všech tří hran jsou získány tři vrcholy sousedních ploch.

Po získání vrcholů sousedních ploch pro každou plochu je provedena první fáze techniky, kde jsou zapsány hodnoty hloubek fragmentů do hloubkového bufferu. K tomuto účelu je vytvořen jednoduchý shader program, kde je důležitý pouze vertex shader. Fragment shader tohoto shader programu je reprezentován prázdnou hlavní funkcí z důvodu jeho nutnosti v shader programu. Vertex shader zpracovává pouze pozice vrcholů, ze kterých jsou interpolovány pozice fragmentů. Zápis do hloubkového bufferu je poté prováděn automaticky. Následující kód představuje hlavní funkci vertex shaderu, zpracovávající pozici.

```
void main()
{
    //in_Position is input attribute
    gl_Position = (projectionMatrix * viewMatrix * modelMatrix) *
                  vec4(in_Position,1.0);
}
```

Před použitím shader programu pro první fázi jsou na straně klienta, tedy aplikace, nastaveny určité parametry pro opakovaný korektní proces vytváření stínů. Jelikož je fragment shader reprezentován prázdnou funkcí, není výsledek barva fragmentu, tudíž není potřeba zápis do barevného bufferu. Barevný buffer je možné určit funkcí *glDrawBuffer*. V této fázi je hlavním účelem získání hloubek, k čemuž je, pomocí funkce *glDepthMask*, povolen zápis do hloubkového bufferu. Defaultně je zápis povolen, nicméně v dalších fázích se povolení mění. Stejně tak se mění nastavení stencil bufferu a stencil testu, což by mohlo ovlivnit zápis hloubek, a proto je, pomocí funkcí *glStencilFunc* a *glStencilOp*, zvoleno vhodné nastavení. Podrobnější popis jednotlivých funkcí je možné nalézt v [2][3]. Vhodné nastavení je obsaženo v následujícím kódu.

```
glDrawBuffer(GL_NONE);
glDepthMask(GL_TRUE);
glStencilFunc(GL_ALWAYS, 0, 0xff);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

V druhé fázi techniky dochází k vytvoření stínových těles a zápisu hodnot do stencil bufferu na základě algoritmu Z-fail. Většina kódu této fáze je obsažena v geometry shaderu vytvořeného shader programu pro druhou fázi. Vertex shader poskytuje pouze pozici v globálním prostoru a další transformace probíhají již v geometry shaderu. I pro tento shader program je fragment shader reprezentován prázdnou funkcí.

Vytváření stínových těles se provádí v geometry shaderu a využívá se informace o třetích vrcholech sousedních ploch. Počáteční hrany stínového tělesa jsou reprezentovány hranami, které jsou sdíleny osvětlenou a neosvětlenou plochou. Je tedy detekována hrana, která je určena vektorovým součinem mezi opačným směrem příchozího paprsku a normálovým vektorem pro obě plochy, sdílející danou hranu. Pro urychlení vytváření je detekce hran provedena, pouze pokud je hlavní trojúhelník osvětlen. Následující kód představuje způsob detekce požadované hrany.

```
vec3 e1 = ex_WorldPos[2] - ex_WorldPos[0];
vec3 e2 = ex_WorldPos[1] - ex_WorldPos[0];
vec3 e3 = ex_WorldPos[4] - ex_WorldPos[2];
// e4, e5, e6

vec3 normal = normalize(cross(e1, e3));
vec3 lightDirection = normalize(ex_WorldPos[0] - lightPosition);

if(dot(normal, -lightDirection) > 0.0)
{
    // 1st neighboring triangle
    normal = normalize(cross(e2, e1));

    if(dot(normal, -lightDirection) <= 0.0)
    {
        EmitVolumeFace(ex_WorldPos[0], ex_WorldPos[2]); // create face
    }
    // 2nd & 3rd triangle - same code with another edges and vertices
}
```

Detekovaná hrana je následně použita pro vytvoření plochy pláště stínového tělesa, která je protažena do nekonečna. Protažení do nekonečna je způsobeno nastavením hodnoty W, komponenty homogenních souřadnic, na hodnotu 0, což při perspektivním dělení způsobí požadovaný efekt.

```
//edgePoint1 and edgePoint2 are points of detected edge
vec3 lightDirection = normalize(edgePoint1 - lightPosition);

gl_Position = projectionMatrix * viewMatrix * vec4((edgePoint1), 1.0);
EmitVertex();

gl_Position = projectionMatrix * viewMatrix * vec4(lightDirection, 0.0);
EmitVertex();

lightDirection = normalize(edgePoint2 - lightPosition);

gl_Position = projectionMatrix * viewMatrix * vec4((edgePoint2), 1.0);
EmitVertex();

gl_Position = projectionMatrix * viewMatrix * vec4(lightDirection, 0.0);
EmitVertex();

EndPrimitive();
```

Z důvodu použití algoritmu Z-fail pro tuto ukázkovou implementaci je nutné uzavřít stínové těleso předním a zadním uzávěrem. Přední uzávěr je tvořen všemi hlavními trojúhelníky, které jsou osvětleny. Zadní uzávěr je poté tvořen stejnými trojúhelníky, které jsou však, ve směru svitu světla, posunuty do nekonečna. Následující části kódu geometry shaderu představují vytvoření zadního a předního uzávěru stínového tělesa.

```
//Back cap
lightDirection = normalize(ex_WorldPos[2] - lightPosition);
gl_Position = projectionMatrix * viewMatrix * vec4(lightDirection, 0.0);
EmitVertex();

lightDirection = normalize(ex_WorldPos[0] - lightPosition);
gl_Position = projectionMatrix * viewMatrix * vec4(lightDirection, 0.0);
EmitVertex();

lightDirection = normalize(ex_WorldPos[4] - lightPosition);
gl_Position = projectionMatrix * viewMatrix * vec4(lightDirection, 0.0);
EmitVertex();

EndPrimitive();
```

```
//Front cap
gl_Position = projectionMatrix * viewMatrix * vec4(ex_WorldPos[0], 1.0);
EmitVertex();

gl_Position = projectionMatrix * viewMatrix * vec4(ex_WorldPos[2], 1.0);
EmitVertex();

gl_Position = projectionMatrix * viewMatrix * vec4(ex_WorldPos[4], 1.0);
EmitVertex();

EndPrimitive();
```

Stejně jako u první fáze, i před použitím fáze druhé je nutné nastavit určité parametry. Zápis do hloubkového bufferu je zakázán, neboť nesmí dojít k přepsání hodnot z první fáze. Pouze dochází k hloubkovému testování mezi hloubkami fragmentů stínových těles a hloubkami uložených v hloubkovém bufferu. Pro výsledky těchto testů jsou specifikovány určité operace nad hodnotami stencil bufferu. Tyto operace odpovídají právě algoritmu Z-fail. Jelikož je nutné specifikovat rozdílné operace pro fragmenty předních a zadních ploch stínového tělesa, je v OpenGL definována funkce *glStencilOpSeparate*, se kterou je to možné. S nutností zpracovat přední i zadní plochy souvisí i deaktivace cull-face módu. Níže uvedený kód obsahuje požadované nastavení.

```
glDrawBuffer(GL_NONE);
glDepthMask(GL_FALSE);
glDisable(GL_CULL_FACE);
glStencilFunc(GL_ALWAYS, 0, 0xff);
glStencilOpSeparate(GL_FRONT, GL_KEEP, GL_DECR_WRAP, GL_KEEP);
glStencilOpSeparate(GL_BACK, GL_KEEP, GL_INCR_WRAP, GL_KEEP);
```

Při použití algoritmu Z-fail může docházet k problému, kdy je ve fázi Clipping odstraněn zadní uzávěr stínového tělesa z důvodu větší vzdálenosti, než je vzdálenost zadní ořezávací roviny. Řešení tohoto problému je přímo v OpenGL, které nabízí funkcionalitu *Depth clamping* [2]. Po aktivaci této funkcionality dochází k transformaci vzdáleností větších, než je vzdálenost zadní ořezávací roviny, na maximální vzdálenost, danou právě touto rovinou.

```
glEnable(GL_DEPTH_CLAMP);
```

Ve třetí a také poslední fázi této techniky je využit shader program pro výpočet osvětlení. Výsledná scéna je vykreslena dvakrát, přičemž je pokaždé jiné nastavení stencil testu. V prvním vykreslení jsou vykresleny pouze fragmenty, které mají být vykresleny na pozici, pro kterou je hodnota stencil bufferu rovna hodnotě 0. Tyto fragmenty nejsou ve stínu a je pro ně vypočítána výsledná intenzita osvětlení na základě všech tří složek. Oproti tomu ve druhém vykreslení jsou vykresleny fragmenty na pozicích, pro které je hodnota stencil bufferu jiná než hodnota 0. Pro tyto fragmenty je vypočítána výsledná intenzita pouze na základě ambientní složky. Hodnoty stencil bufferu při této fázi nesmí být měněny.

Pro obě vykreslení je specifikován barevný buffer jako zadní barevný buffer defaultního framebufferu. Stejně tak je povolen zápis do hloubkového bufferu, který je nastaven na původní hodnoty z důvodu možných chyb při hloubkovém testování, i přestože uložené hloubky odpovídají hloubkám nejbližších fragmentů právě této scény.

```
glDrawBuffer(GL_BACK);
glDepthMask(GL_TRUE);
glClear(GL_DEPTH_BUFFER_BIT);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

glStencilFunc(GL_EQUAL, 0x0, 0xff);
/*Render unshadowed part*/

glStencilFunc(GL_NOTEQUAL, 0x0, 0xff);
/*Render shadowed part (only ambient)*/
```

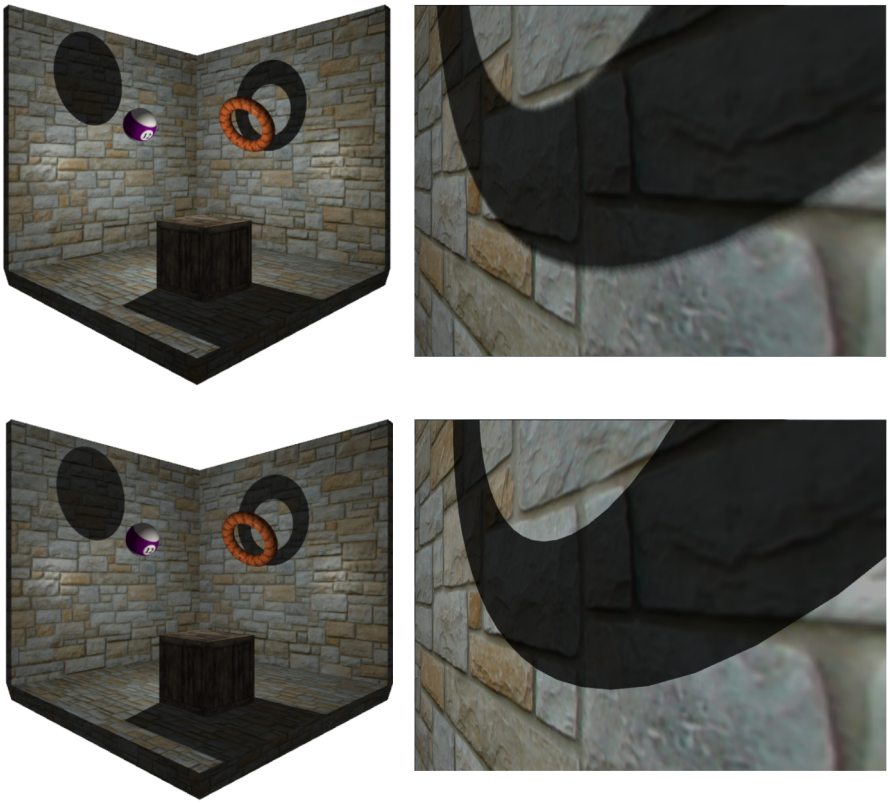
I při použití této techniky mohou vznikat různé nežádoucí jevy. Základní nežádoucí jevy, se kterými je možné se setkat v této implementaci, a jejich možná řešení jsou popsány v teoretické příloze.

7.3 Srovnání technik Shadow mapping a Stencil shadow volume

Ze způsobu implementací obou technik a vzhledu výsledných stínů, které je možné vidět na obrázku 7.5, lze odvodit základní výhody a nevýhody, na základě kterých je, mezi těmito technikami, rozhodováno.

Základní výhodou techniky Shadow mapping je jednodušší implementace pro všechny tři typy světelného zdroje, což je možné odvodit z předchozí implementace pro bodové světlo, která je z těchto tří nejsložitější, a i přesto není příliš náročná. Další výhodou je jednodušší možnost vytváření takzvaných měkkých stínu (angl. soft shadows), což znamená, že hrany stínů nejsou příliš ostré. Jako nevýhody této techniky je možné považovat množství nežádoucích jevů, převážně vznikající aliasing. Nevýhodná se také může jevit nutnost vlastní shadow mapy pro každé světlo ve scéně, s čímž souvisí i nutnost vykreslení scény z pozice každého světla.

Viditelnou výhodou techniky Stencil shadow volume je nevznikající aliasing, neboť pro každý fragment výsledné scény je určena jedna hodnota ve stencil bufferu. Při použití více světel zůstává, ve většině případů, počet vykreslení stále stejný, což je způsobeno možností vytvořit všechna stínová tělesa v geometry shaderu během jednoho vykreslení. Také je využit stále jeden stencil buffer. Nevýhodou je pak složitější implementace, způsobena vytvářením stínových těles, a nutnost uzavřených objektů z důvodu potřebné informace o sousedních plochách. Nevýhodou také mohou být ostré stíny, které jsou touto technikou vytvářeny. Měkkých stínů lze však také určitým způsobem dosáhnout, například pomocí několika světel uspořádaných blízko sebe.



Obrázek 7.5: Výsledné zobrazení stínů obou technik

I přestože výsledné stíny těchto implementací neodpovídají přesně reálnému vzhledu stínů, je použití těchto technik pro real-time aplikace velice dostačující.

Výsledné a spustitelné implementace obou technik jsou obsaženy v příloze této práce, v následujících uvedených složkách.

Shadow mapping - /Examples/06. Shadow mapping - point light

Stencil shadow volume - /Examples/07. Stencil shadow volume - point light

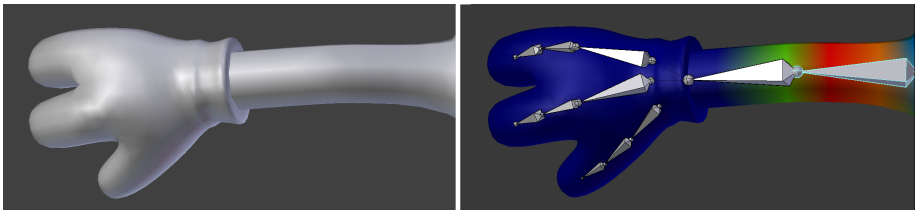
Kapitola 8

Animace

Z hlediska vizuálních efektů patří animace k důležitým prvkům, co se týče dynamičnosti scény. V určitých aplikacích tvoří animace nedílnou součást výsledné vizualizace, v jiných jsou zase použity pouze pro dosažení realističtějšího dojmu. Obecně animace je možné dělit na následující dva typy.

První typ představuje animace, které je možno provádět plynulou změnou modelové matice příslušného objektu. Jedná se o jednoduché transformace, jako jsou posuv, rotace nebo změna měřítka. Jednoduchým příkladem tohoto typu animací je animace výtahové kabiny, kde plynulá změna pozice vytváří dojem jízdy směrem nahoru nebo dolů. Aplikací modelové matice, která představuje transformaci, jsou však ovlivněny všechny vrcholy objektu, což je jistá nevýhoda tohoto typu animace.

Druhý typ animací slouží pro objekty, jejichž animace je složitější a nelze provést pouhou změnou modelové matice jako u prvního typu. Příklady takových objektů jsou lidské postavy, zvířata nebo podobné objekty. Důležitou vlastností tohoto typu animace je možnost aplikace transformace pouze na určitou část objektu. Tuto možnost zajišťují takzvané kosti (angl. bones), které mohou dohromady vytvářet kostru neboli armaturu celého objektu. Každá kost poté může definovat určitou transformaci, která ovlivňuje vrcholy na základě vah (angl. weights). Váhy, které jsou definovány pro každý vrchol, určují, jak moc je daný vrchol ovlivněn transformacemi kostí, na kterých je závislý. Tento typ animace nicméně vyžaduje práci animátora, který danému objektu vytvoří kostru a jednotlivým vrcholům vhodně přiřadí váhy. Následně jsou vytvořeny takzvané klíčové snímky, které určují transformace kostí pro konkrétní čas, a které dohromady vytváří výslednou animaci objektu. Práce animátora je, ve většině případů, prováděna v externích grafických programech (Blender, Maya, 3D Studio Max aj.), ze kterých je objekt, včetně definované animace, exportován do souborů daného typu, který uložení animace podporuje (.dae, .MD5 aj.). Úkolem programátora je poté tato data přečíst a vhodně zpracovat.

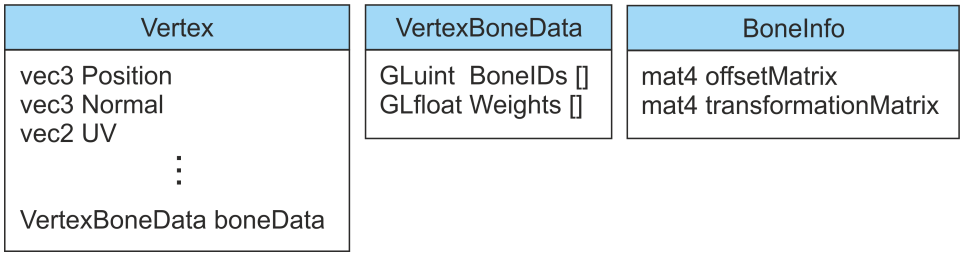


Obrázek 8.1: Rozložení vah pro část objektu (Blender)

8.1 Implementace animace

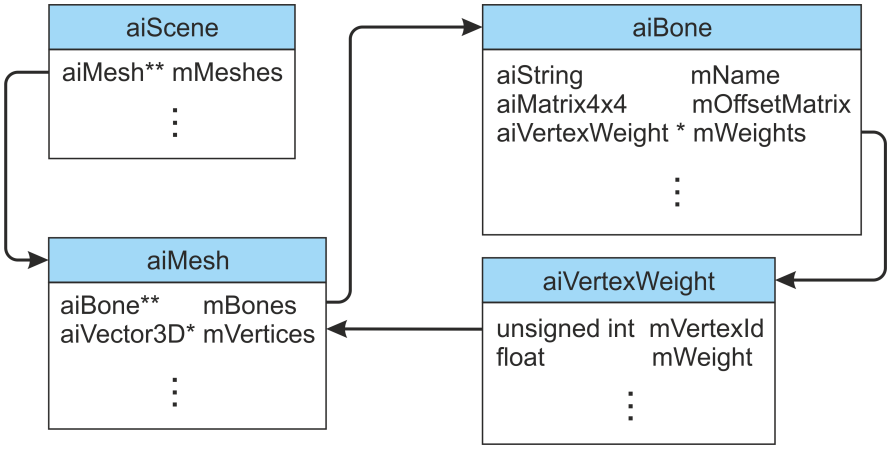
V této části kapitoly je představen základní princip zpracování dat objektu a jejich využití pro animaci v aplikaci využívající OpenGL. Pro čtení souboru je opět využita knihovna Assimp [4], která poskytuje požadovaná data pro animaci ve vhodných strukturách, které budou během popisu hlavních částí implementace zmíněny.

Základní úpravou je rozšíření dat vrcholu o položky, reprezentující indexy kostí, které daný vrchol mohou ovlivnit, a také příslušné váhy. V této implementaci je možné, aby byl vrchol ovlivněn maximálně pěti kostmi, což je ve většině případů dostačující. Indexy kostí a váhy pro jednotlivé vrcholy jsou obsaženy v pomocné struktuře, která je součástí struktury vrcholu. Mimo to je také vytvořena struktura reprezentující jednotlivé kosti, která obsahuje jejich offset matici a transformaci.



Obrázek 8.2: Struktury pro uložení požadovaných dat

Získání dat pro první dvě struktury, zobrazené na obrázku 8.2, je cíl první části implementace. Po načtení souboru jsou jednotlivé objekty scény uloženy ve struktuře aiMesh, která obsahuje pole kostí, ovlivňujících tento objekt. Jednotlivé kosti, reprezentované strukturou aiBone, obsahují pole struktur aiVertexWeight, kde každá struktura obsahuje index vrcholu a jeho váhu pro danou kost. [4]



Obrázek 8.3: Propojení struktur

Následující kód představuje zpracování kostí jednoho objektu, kde jsou jména jednotlivých kostí mapována na příslušný index, a ke každé kosti je vytvořena struktura pro uložení matic. Do této struktury je následně uložena offset matice, která provádí transformaci z prostoru objektu do prostoru dané kosti [4]. Nakonec jsou zpracovány struktury aiVertexWeight dané kosti, kde

do připraveného pole, jehož velikost je stejná jako počet vrcholů objektu, je pro daný vrchol uložen index zpracovávané kosti a příslušná váha. Toto pole poté slouží k inicializaci výsledné struktury vrcholu.

```
GLuint boneIndex;

for(GLuint i = 0; i < mesh->mNumBones; i++)
{
    std::string boneName = (mesh->mBones[i])->mName.data;

    if(boneToIndexMap.find(boneName) == boneToIndexMap.end())
    {
        boneIndex = numOfBones;
        boneToIndexMap[boneName] = boneIndex; //creating new bone index
        numOfBones++;

        BoneInfo emptyInfo;
        boneInfo.push_back(emptyInfo); //new structure for bone matrices
    }
    else
    {
        boneIndex = boneToIndexMap[boneName]; //getting existing bone index
    }

    boneInfo[boneIndex].offsetMatrix = mesh->mBones[i]->mOffsetMatrix;

    //Getting weights for vertices affected by this bone
    for(GLuint j = 0; j < mesh->mBones[i]->mNumWeights; j++)
    {
        GLuint vertexID = mesh->mBones[i]->mWeights[j].mVertexId;
        GLfloat boneWeight = mesh->mBones[i]->mWeights[j].mWeight;
        bones[vertexID].Add(boneIndex, boneWeight);
    }
}
```

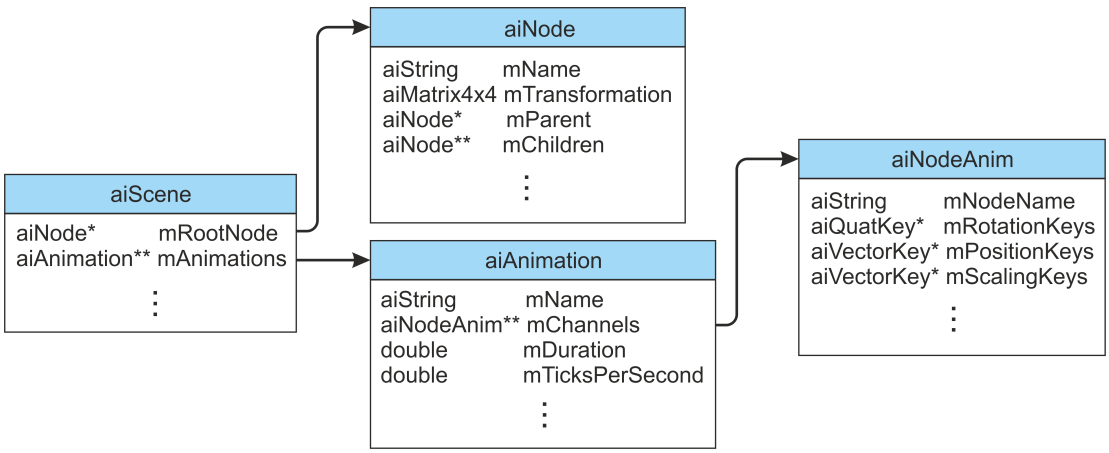
Po zpracování potřebných dat pro jednotlivé vrcholy objektu následuje další část implementace, kde jsou vypočítávány transformace kostí v daném čase animace. Tyto transformace jsou nakonec poskytnuty vertex shaderu, kde jsou aplikovány.

Načtená scéna může obsahovat několik různých animací (např. chůze, běh, skok aj.), které jsou reprezentovány strukturou aiAnimation. Tyto struktury jsou uloženy v poli struktury aiScene. Struktura aiAnimation obsahuje základní informace o animaci (název, délka, framerate atd.) a také pole struktur aiNodeAnim, které představují kosti nutné pro tuto animaci. Struktura aiNodeAnim obsahuje název kosti a jednotlivé údaje o pozicích, rotacích a změnách velikostí této kosti v daném čase animace [4]. Zpravidla se nedefinují tyto údaje pro každý snímek, a proto je nutné, pokud se animace nachází v čase, pro který nejsou tyto údaje definovány, tyto údaje interpolovat. Následující kód ukazuje způsob interpolace rotace kosti, k čemuž je využito takzvaných kvaternionů. Základní informace o kvaternionech jsou uvedeny v teoretické příloze.

```
if(nodeAnim->mNumRotationKeys == 1) //if there's only one key
{
    return nodeAnim->mRotationKeys[0].mValue;
}
GLuint index = FindRotation(animationTime, nodeAnim);
GLuint nextIndex = index + 1;

//Getting interpolation factor from interval [0,1]
GLfloat keyTime = (GLfloat) nodeAnim->mRotationKeys[index].mTime;
GLfloat nextKeyTime = (GLfloat) nodeAnim->mRotationKeys[nextIndex].mTime;
GLfloat deltaTime = nextKeyTime - keyTime;
GLfloat factor = (animationTime - keyTime) / deltaTime;

//Quaternion interpolation
aiQuaternion quaternion;
aiQuaternion start = nodeAnim->mRotationKeys[index].mValue;
aiQuaternion end = nodeAnim->mRotationKeys[nextIndex].mValue;
aiQuaternion::Interpolate(quaternion, start, end, factor);
return quaternion.Normalize();
```

Výsledkem každé interpolace posuvu, rotace nebo změny měřítka je transformační matice. Všechny tři transformace jsou následně složeny do jedné, která však stále neodpovídá výsledné transformaci kosti. Často se při tvorbě kostry objektu tvoří i závislosti mezi jednotlivými kostmi. Definované závislosti způsobují přenos transformace rodičovské kosti na kosti, které jsou na ní závislé.

Knihovna Asssimp organizuje položky načtené scény (např. světla, kamera, objekty, kosti atd.) do stromové struktury na základě závislostí mezi nimi. Každá položka je reprezentována strukturou aiNode, která obsahuje jedinečné jméno, transformační matici, odkazy na nadřazený uzel a potomky, aj. Je tedy rekurzivně procházena tato stromová struktura a zjišťováno, zda název struktury aiNode je ekvivalentní s názvem jakékoliv struktury aiNodeAnim dané animace. V případě rovnosti představuje daná struktura aiNode položku kosti a je provedena interpolace pro získání transformace.



Obrázek 8.4: Propojení struktur

Následující kód představuje porovnávání názvu struktury aiNode s názvy struktur aiNodeAnim dané animace, kde při shodě je navrácena struktura aiNodeAnim, obsahující klíčové údaje pro následnou interpolaci transformací.

```
for(GLuint i = 0; i < animation->mNumChannels; i++)
{
    aiNodeAnim* nodeAnim = animation->mChannels[i];

    if(std::string(nodeAnim->mNodeName.data) == nodeName)
    {
        return nodeAnim;
    }
}
```

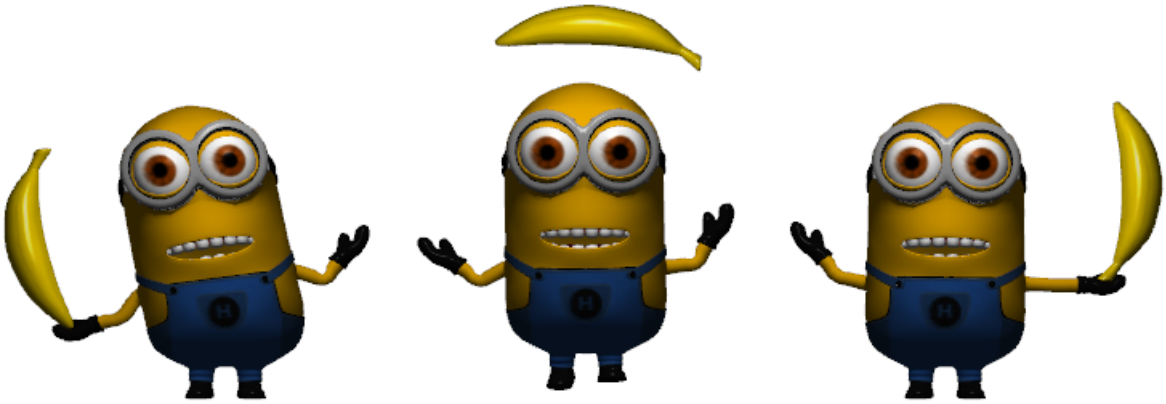
U položek, které nejsou kostmi, zůstává transformační matice ze struktury aiNode beze změny, a pokud je tato položka závislá na nadřazené položce a zároveň je nadřazenou položkou pro další položky, je tato transformační matice vynásobena maticí nadřazené položky a poskytnuta všem potomkům, což vytváří řetězec transformací.

V případě položky, reprezentující kost, se výsledná transformace odvíjí od transformace nadřazené kosti, vlastní interpolované transformace a transformace reprezentované offset maticí. Na celou transformaci je poté aplikována inverzní transformace kořene stromu, což je jedna z věcí, kterou je nutné provést pro korektní transformaci v případě použití knihovny Assimp. Následující kód představuje skládání výsledné transformační matice kosti.

```
GLuint boneIndex = boneToIndexMap[nodeName];

/*
rootInverseMatrix - root inverse transformation matrix
nodeTransformation - interpolated transformation with parent transformation
*/

boneInfo[boneIndex].transformationMatrix =
                                        rootInverseMatrix *
                                        nodeTransformation *
                                        boneInfo[boneIndex].offsetMatrix;
```

Po dokončení průchodu stromovou strukturou uzlů aiNode jsou v příslušných strukturách, které drží informaci o transformaci a offset matici konkrétní kosti, uloženy konečné transformační matice pro daný čas animace. Tyto konečné transformace jsou poskytnuty vertex shaderu. Na základě indexů kostí, které jsou definované pro každý vrchol, jsou z těchto transformací vybírány transformace, které daný vrchol ovlivňují. Pomocí vah je poté regulována síla jednotlivých transformací. Část vertex shaderu, která provádí aplikaci transformace kostí na pozici vrcholu, je obsažena v následujícím kódu.

```
uniform mat4 boneTransform[MAX_BONES];

void main(void)
{
    //in_Position, in_BoneIDs[], in_Weights[] are input attributes

    mat4 finalBoneTransform = boneTransform[in_BoneIDs[0]] * in_Weights[0];

    for(uint i = 1; i < 5; i++)
    {
        finalBoneTransform += boneTransform[in_BoneIDs[i]] * in_Weights[i];
    }

    vec4 transformedPosition = finalBoneTransform * vec4(in_Position, 1.0);
    gl_Position = (projectionMatrix * viewMatrix * modelMatrix) *
                  transformedPosition;
}
```



Obrázek 8.5: Výsledná animace

Výsledná animace dodává grafické scéně dynamičnost, která je u scény se statickými objekty postrádána. Nicméně neustálá interpolace transformací pro všechny kosti, která je navíc prováděna jednotkou procesoru, způsobuje výrazné snížení počtu vykreslených snímků za vteřinu.

Spustitelná aplikace pro tuto kapitolu je obsažena v příloze, ve složce /Examples/08. Animation .

Kapitola 9

Částicový systém

Částicový systém je obecný pojem pro způsob simulace různých jevů. Často je využívám pro simulaci reálných přírodních jevů, jako jsou kouř, oheň, déšť a mnoho dalších, nicméně má využití i pro simulaci jevů nereálných pro účel různých vizuálních efektů. Způsob řešení simulace těchto jevů je založen na určitém počtu částic, které svým chováním reprezentují konkrétní jev. Částice je většinou reprezentovaná základním primitivem, jako je bod, trojúhelník, čtyřúhelník apod. Simulace konkrétního jevu je závislá převážně na způsobu pohybu částic, který může být pro jednotlivé částice vypočítáván jednoduchými až velmi komplexními matematickými výpočty.

9.1 Transform feedback

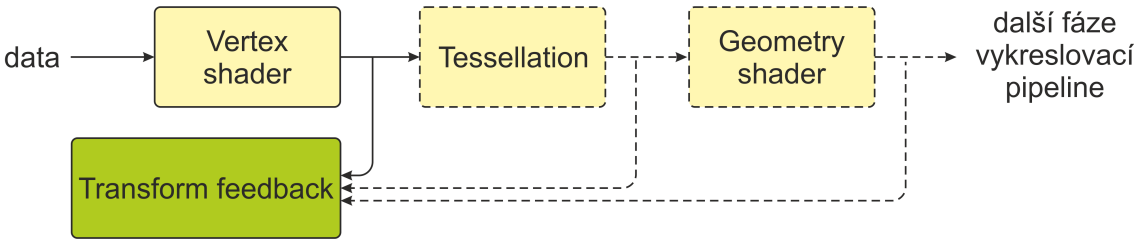
Ve starších verzích OpenGL probíhal způsob simulace těchto jevů na základě častého přenosu dat ze strany klienta na grafickou kartu. Na straně aplikace docházelo k určité transformaci částic tak, aby odpovídaly chování simulovaného jevu, a následně byla tato data poskytnuta grafické kartě pro vykreslení. Přenos dat na grafickou kartu byl neustále opakován pro každý jednotlivý snímek, což způsobovalo velkou časovou náročnost, která limitovala použití většího počtu částic.

Z tohoto důvodu byla pro OpenGL představena funkcionalita s názvem *Transform feedback* [3][2]. Jedná se o proces zachytávání určitých hodnot, které nejčastěji odpovídají vlastnostem vrcholů, tvořících částice, do příslušných asociovaných bufferů z poslední fáze grafické pipeline, která zpracovává vrcholy. Poslední fází, zpracovávající vrcholy, se rozumí programovatelná fáze pomocí vertex shaderu nebo geometry shaderu, pokud je obsažen. Od OpenGL verze 4.0 je možné zachytávat hodnoty i z nové fáze vykreslovací pipeline, teselační fáze, která je předmětem kapitoly Teselace. Výpočet a aplikace transformací pro jednotlivé vrcholy je možné provádět v těchto fázích, což řeší problém častého přenosu dat na grafickou kartu a navíc urychluje výpočet z důvodu paralelizace výpočtů na grafické kartě.

K využití této funkcionality je v OpenGL definován takzvaný transform feedback objekt, což je objekt, se kterým je asociován jeden nebo více bufferů pro ukládání zachycených vlastností vrcholů. Možnost asociace více bufferů s tímto objektem je z důvodu možného rozložení zachycených hodnot do různých bufferů. Ve starších verzích OpenGL bylo možné ukládat všechny zachycené hodnoty pouze společně do jednoho bufferu nebo naopak každou do samostatného bufferu. Od OpenGL verze 4.0 je možné specifikovat hodnoty, které se budou ukládat společně

do jednoho bufferu a které do bufferů dalších. [3][2]

Buffery, asociované s transform feedback objektem, jsou reprezentovány array buffery, které jsou pro tento objekt označeny jako cílová úložiště. Tyto buffery, obsahující zachycená data, je poté možno použít jako zdroj dat pro následující vykreslení transformovaných částic.



Obrázek 9.1: Umístění transform feedback funkcionality ve vykreslovací pipeline

9.2 Implementace částicového systému

Předmětem následující ukázkové implementace je vytvoření efektu ohně, který je založen na vlastním jednoduchém algoritmu. Tento algoritmus využívá základní goniometrické funkce.

V této implementaci je transformace prováděna s body, kde každý bod reprezentuje částici. Při vykreslení jsou však z těchto bodů, pomocí dvou trojúhelníků, vytvořeny čtyřúhelníky. Důvodem takové změny je možnost mapovat texturu na danou částici, a tím ji určit libovolný vzhled namísto jednoduchého bodu. Pro vytvoření efektu ohně musí každá částice nabývat určitých vlastností, pro jejichž sjednocení je vytvořena struktura.

Particle	
vec3	position
GLuint	type
GLfloat	speed
GLfloat	life
GLfloat	maxLife
GLfloat	amplitude
GLfloat	rotation
GLfloat	angle

Obrázek 9.2: Struktura částice

Při použití *Transform feedback* funkcionality se zpravidla používají dva transform feedback objekty, kde je s každým asociován jeden array buffer pro ukládání zachycených hodnot. Důvodem je nutnost současného čtení a zápisu z/do array bufferu při fázi transformace, což by při použití jednoho array bufferu vedlo k nechtěným výsledkům.

Oba buffery, asociované s transform feedback objekty, jsou inicializovány daty, která reprezentují pole částic o vhodné velikosti, určené maximálním možným počtem vygenerovaných částic. Jde však pouze o alokaci paměti pro oba buffery, neboť data pro jednotlivé částice jsou generována v geometry shaderu příslušného shader programu. Pouze první částice vytvořeného pole je inicializovaná daty, které ji identifikují jako zdroj všech částic. Následující kód představuje vytvoření požadovaných objektů a vytvoření asociace mezi nimi.

```
glGenTransformFeedbacks(2, TFO); //new IDs for transform feedback objects
glGenBuffers(2, VBO); //new IDs for buffers

for(GLuint i = 0; i < 2; i++)
{
 glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, TFO[i]);
 glBindBuffer(GL_ARRAY_BUFFER, VBO[i]);
 glBufferData(GL_ARRAY_BUFFER, sizeof(particles), particles ,GL_DYNAMIC_DRAW);
 glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, VBO[i]); //association
}
```

Pro transformaci částic je vytvořen shader program, který obsahuje vertex shader a geometry shader, ve kterém je možné generovat nové částice a ze kterého jsou zachytávány požadované hodnoty. Vertex shader pouze předá vstupní hodnoty další fázi vykreslovací pipeline. Názvy proměnných, jejichž hodnoty mají být zachyceny, jsou specifikovány pomocí funkce *glTransformFeedbackVaryings* [2][3]. Po provedení specifikace je znovu provedeno linkování shader programu, což je nutné u většiny grafických karet. Specifikace názvů proměnných a způsob ukládání hodnot představuje následující kód.

```
const GLchar* varyings[] = { "Position","Type","Speed","Life",
                             "MaxLife","Amplitude","Rotation","Angle" };

glTransformFeedbackVaryings(program, sizeof(varyings)/sizeof(GLchar*),
                            varyings, GL_INTERLEAVED_ATTRIBS);
glLinkProgram(program);
```

Funkce geometry shaderu je závislá na typu částice, kterou aktuálně zpracovává. Pokud se jedná o částici, která je zdrojem částic, zjišťuje se překročení její maximální životnosti. Pokud je maximální životnost překročena, je vygenerován určitý počet nových částic s náhodnými hodnotami pro určité vlastnosti. Stejně tak je aktualizována životnost zdroje částic na počáteční hodnotu pro možné opakování. Pokud hodnota překročena není, je pouze aktualizována životnost zdroje částic o určitý krok. Částice tohoto typu by neměla nikdy zaniknout.

V případě jiného typu částice je proveden algoritmus, který simuluje pohyb částic pro efekt ohně a modifikuje určité vlastnosti pro příští vykreslení. Tento algoritmus pracuje na principu pohybu částic po sinusoidě s plynulou změnou amplitudy a je proveden, pouze pokud hodnota životnosti zpracovávané částice je nižší než hodnota její maximální životnosti. Životnost částice je závislá na své rychlosti, neboť čím vyšší je hodnota rychlosti částice, tím rychleji také stárne. Podobným způsobem je provedena i plynulá změna amplitudy, kde čím větší je amplituda, tím větší je hodnota o kterou se sníží. Určité vlastnosti částice se nemění, jelikož slouží pouze jako hodnoty pro výpočet předchozích uvedených vlastností. Použití tohoto algoritmu znázorňuje následující část kódu geometry shaderu.

```
//ex_{attributeName}[]  are input attributes
//Position, Type, Speed, Life, MaxLife, Amplitude, Rotation, Angle

//{attributeName} are output attributes
//Position, Type, Speed, Life, MaxLife, Amplitude, Rotation, Angle

float currentLife = ex_Life[0] + deltaTime/1000 * ex_Speed[0];

if(currentLife < ex_MaxLife[0])
{
    vec3 moveVector = vec3(cos(radians(ex_Rotation[0])), 0.0,
                           sin(radians(ex_Rotation[0])));
    moveVector *= ex_Amplitude[0] * sin(radians(ex_Angle[0]));
    vec3 upVector = vec3(0.0, ex_Position[0].y - origin.y , 0.0);
    upVector.y += ex_Speed[0] * deltaTime/1000;
    Position = origin + moveVector + upVector; //origin = source position
    Amplitude = ex_Amplitude[0] *
                (1 - deltaTime/GetRandomNumber(1000, 5000, 0));
    Amplitude = max(Amplitude,0.2);
    Angle = ex_Angle[0] + deltaTime/10;
    Angle = mod(Angle, 360.0);
    Life = currentLife;
    /*Fill remaining output attributes with input attributes */

    EmitVertex();
    EndPrimitive();
}
```

Shader program, který využívá *Transform feedback*, může tvořit výjimku, kdy nemusí být obsažen povinný fragment shader z důvodu možného zamezení procesu rasterizace. Zamezení lze provést funkcí *glEnable* s konstantou *GL_RASTERIZER_DISCARD*. V takovém případě dochází k ukončení vykreslovacího procesu před rasterizací, což zapříčiňuje zbytečnost fragment shaderu.

Před použitím tohoto shader programu je jeden z array bufferů aktivován jako zdroj dat pro vstupní parametry vertex shaderu a druhý je aktivován jako úložiště zachycených dat pomocí aktivace transform feedback objektu, se kterým je asociován.

Se zachytáváním dat nutně souvisí i dvě funkce, které signalizují začátek a konec zachytávání z toho důvodu, že ne všechny shader programy tuto funkcionalitu využívají. Jedná se o funkce *glBeginTransformFeedback* a *glEndTransformFeedback*, kde pro první funkci je specifikován typ primitiva, který je stejný jako typ výstupního primitiva geometry shaderu [2][3]. Při generování nových částic pomocí geometry shaderu je obtížné sledovat počet částic, které jsou obsaženy v bufferech. Z toho důvodu je využita funkce *glDrawTransformFeedback* [2][3], která automaticky zjistí počet částic z transform feedback objektu po jeho posledním použití. Je tedy použit i neaktivní transform feedback objekt, se kterým je asociován buffer, sloužící jako aktuální zdroj dat. Toto nastavení a způsob vykreslení je znázorněn v následující části kódu.

```
glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, TFO[currentTF]);
glEnable(GL_RASTERIZER_DISCARD);
transformProgram->Enable(); //activating shader program
/*Setting uniform variable values*/

glBeginTransformFeedback(GL_POINTS);
    if(isFirst)
    {
        glDrawArrays(GL_POINTS, 0, 1);
        isFirst = false;
    }
    else
    {
        glDrawTransformFeedback(GL_POINTS, TFO[currentVB]);
    }
glEndTransformFeedback();
```

Pro fázi vykreslení je vytvořen shader program, který obsahuje vertex, geometry i fragment shader. V geometry shaderu jsou pomocí techniky Billboarding ze vstupních bodů vytvořeny triangulované čtyřúhelníky. Tato technika zamezuje situaci, kdy nejsou čtyřúhelníky při vyšších pozorovacích úhlech příliš dobře vidět. Principem této techniky je vytvoření vrcholů čtyřúhelníka posunem po vektorech, které jsou závislé na pozici pozorovatele. Velikost čtyřúhelníků je poté určena velikostí tohoto posunu, která se v této implementaci snižuje na základě rozdílu aktuální životnosti od maximální životnosti částice. Vytvořeným vrcholům jsou přiřazeny texturovací souřadnice pro možnost texturování ve fragment shaderu, které je rovněž ovlivněno životností částice. Následující část kódu představuje vytvoření jednoho ze čtyř vrcholů v geometry shaderu.

```
void main(void)
{
    vec3 position = gl_in[0].gl_Position.xyz; //particle position
    vec3 toCamera = normalize(cameraPosition - position);
    vec3 up = normalize(cameraUp);
    vec3 side = normalize(cross(toCamera, up));
    float sizeCoeff = (GS_MaxLife[0] - GS_Life[0]) / GS_MaxLife[0];
    float modSize =  size * clamp(sizeCoeff, 0.0, 1.0);

    //1st vertex
    position -= side * modSize;
    position -= up * modSize;
    gl_Position = projectionMatrix * viewMatrix * vec4(position, 1.0);
    FS_UV = vec2(1.0, 0.0);
    FS_Life = GS_Life[0]; // particle life
    FS_MaxLife = GS_MaxLife[0]; //particle max life
    EmitVertex();

    //2nd,3rd & 4th vertex with different position calculation and UV */
    EndPrimitive();
}
```

Před použitím tohoto shader programu je array buffer, do kterého byla zachycena data v předchozí části, použit pro vykreslení, které je provedeno stejným způsobem, bez znalosti počtu částic.

Často je při texturování jednotlivých částic využívána OpenGL funkcionalita nazývaná *Blending*, která umožňuje míchání barev na základě různých parametrů. V této implementaci je míchání prováděno na základě alfa kanálu mapované textury. Pro korektní výsledky *Blendingu* je zamezeno zápisu do hloubkového bufferu, což způsobí, že hloubky fragmentů částic jsou testovány s hloubkami, které byly do hloubkového bufferu uloženy před vykreslením částic. Tento způsob však nutí vykreslovat částice jako poslední, kdy je hloubkový buffer naplněn hodnotami z hloubkových testů fragmentů ostatních objektů ve scéně. Toto nastavení a vykreslení představuje následující kód.

```
billboardProgram->Enable();
/* Setting uniform variable values */

glDepthMask(GL_FALSE);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture); //particle texture activation

glDrawTransformFeedback(GL_POINTS, TFO[currentTF]);
```

Obrázek 9.3: Výsledný efekt ohně

Výsledný efekt ohně není příliš realistický, neboť pro simulaci reálného ohně je potřeba komplexnějšího algoritmu. Pro některé real-time aplikace však může být tato jednoduchá simulace ohně dostačující.

Spustitelná aplikace je obsažena v příloze této práce, ve složce /Examples/09. Particle system - fire .

Kapitola 10

Teselace

Teselace je proces, při kterém dochází k dělení základního geometrického primitiva na mnoho menších. Tato funkcionalita je v OpenGL dostupná od verze 4.0 a prozatím představuje jednu z největších novinek, které tato verze přinesla. Z důvodu určitého množství nově vygenerovaných vrcholů je možné zvyšovat úroveň detailů trojrozměrných objektů. Výhodou teselace je možnost plynulé změny množství vygenerovaných vrcholů na základě libovolného parametru, například dle vzdálenosti objektu od pozice pozorovatele. Princip změny úrovně detailů objektu, který předcházel teselaci, spočíval v použití několika stejných objektů, pouze s rozdílem různého počtu ploch. Z těchto objektů bylo poté vybíráno podle vzdálenosti od pozorovatele. Tento způsob měl však jisté nevýhody v nutnosti vytváření více objektů, s čímž souvisí i větší spotřeba paměti grafické karty pro jejich uložení. V případě teselace je dostačující objekt s nízkým počtem ploch, jejichž počet může být zvýšen teselací a následně proveden výpočet pro dotváření detailů. Jedním ze způsobu dotváření detailů je použití takzvané výškové textury neboli displacement textury, jejíž texely reprezentují hodnoty výškové odchylky, které jsou použity pro změnu pozice nově vygenerovaných vrcholů.

10.1 Proces teselace

Pro proces teselace v OpenGL byla do vykreslovací pipeline přidána fáze *Tessellation*, která je umístěna mezi fází s vertex shaderem a fází s geometry shaderem. [2]

Teselační fáze je tvořena třemi menšími fázemi, z nichž dvě jsou programovatelné, s čímž souvisí i dva nové druhy shaderů. *Tessellation Control Shader* (zkr. TCS) pro fázi první a *Tessellation Evaluation Shader* (zkr. TES) pro fázi třetí. Druhá fáze s názvem *Primitive Generation* (zkr. PG) je neprogramovatelná, avšak její výpočet je možné ovlivnit různými parametry [2]. Možnosti nastavení této fáze jsou zmíněny v teoretické příloze této práce.

Pro použití teselace je definován speciální typ primitiva, takzvaný patch, který reprezentuje konstanta *GL_PATCHES* [2]. Patch představuje soubor kontrolních bodů, které mohou být použity pro výpočet vlastností vygenerovaných vrcholů. Při vykreslování jsou kontrolní body nezávisle zpracovávány vertex shaderem a následně společně poskytnuty TCS jako vstupní patch. Maximální počet kontrolních bodů pro jeden patch je závislý na grafické kartě.

Účelem TCS je vytvoření výstupního patche, který nemusí být nutně stejný jako patch vstupní, jelikož zde může docházet k různým transformacím a vytváření nových kontrolních

bodů. TCS je pro každý patch spouštěn tolikrát, kolik kontrolních bodů je obsaženo ve výstupním patchi. Navíc TCS poskytuje hodnoty teselačního levelu, které určují míru teselace. Tato fáze však nemusí být nutně obsažena, což způsobí, že kontrolní body zpracované vertex shaderem jsou formovány přímo do výstupního patche bez jakýchkoliv úprav. Hodnoty teselačního levelu jsou poté nastaveny příslušnými funkcemi na straně klienta. [2]

Druhá fáze PG je nezávislá na výstupním patchi TCS, jelikož tato fáze pracuje se třemi předdefinovanými teselačními doménami. Tato fáze obsahuje algoritmus dělení, který se odvíjí od zvolené teselační domény a dalších parametrů, včetně teselačního levelu. Výsledkem tohoto procesu jsou vygenerované vrcholy, kterým jsou přiřazeny normalizované souřadnice, určující pozici uvnitř domény. Podoba souřadnic se odvíjí od zvolené domény, nad kterou byl proveden proces dělení.

Pro každý vygenerovaný vrchol je spuštěn TES, ve kterém je možné přistupovat k souřadnicím aktuálního vrcholu a k výstupnímu patchi, obsahující kontrolní body. TES slouží k přiřazení konečných a požadovaných vlastností vygenerovaným vrcholům s častým využitím kontrolních bodů. Pro dotváření detailů objektu je zde možné použít různé algoritmy nebo výškovou texturu. Zpracované vrcholy poté pokračují do dalších fází vykreslovací pipeline.

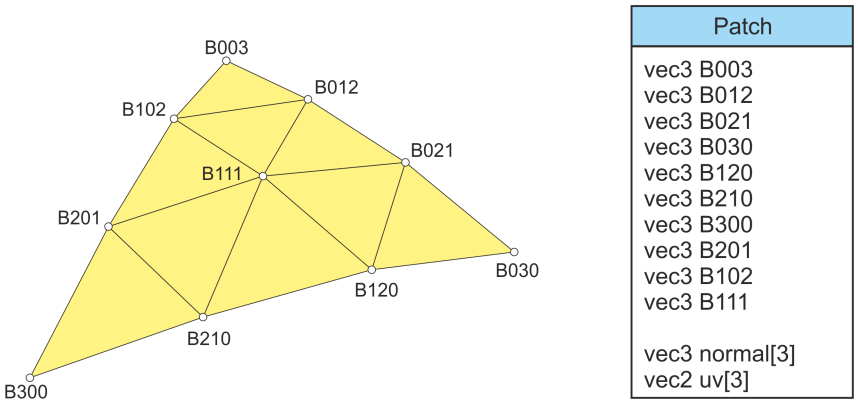
10.2 Implementace PN Triangles

Předmětem následující ukázkové implementace je PN Triangles, což je způsob využití teselace, který byl představen Alexem Vlachosem a jeho kolektivem v roce 2001. Cílem je dosažení zaoblených ploch objektu, k čemuž je využito takzvaného Beziérova trojúhelníka, kterým jsou jednotlivé trojúhelníkové plochy objektu nahrazeny. [14]

Celý proces je převážně obsažen v teselačních shaderech vytvořeného shader programu, nicméně pro korektní použití teselace jsou na straně klienta nastaveny a pozměněny určité věci. Jednou z nich je specifikace primitiva *GL_PATCHES* při vykreslování. Pro toto primitivum není pevně určen počet bodů, a proto je pomocí příkazu *glPatchParameteri* tento počet specifikován. Následující kód představuje použití této funkce s hodnotou 3, neboť vstupní patch je tvořen body původního trojúhelníka.

```
glPatchParameteri(GL_PATCH_VERTICES, 3);
```

Po specifikaci těchto parametrů jsou v TCS přijímány vstupní patche jako soubor tří vrcholů. Tyto body tvoří základ kontrolních bodů Beziérova trojúhelníka, který celkově formuje deset kontrolních bodů. Z toho důvodu je vytvořena struktura, která sjednocuje všechny tyto body a navíc obsahuje normálové vektory a texturovací souřadnice pro pozdější evaluaci vlastností vygenerovaných vrcholů. Tato struktura tvoří výstupní patch TCS.



Obrázek 10.1: Topologie Beziérova trojúhelníka a struktura výstupního patche

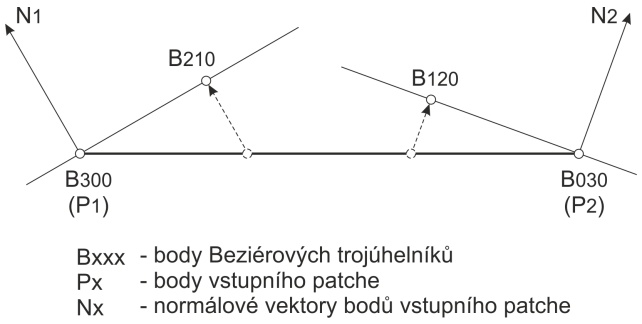
S pomocí kontrolních bodů vstupního patche, které tvoří body B_{300}, B_{030} a B_{003}, jsou vytvořeny zbývající kontrolní body Beziérova trojúhelníka. Každá hrana trojúhelníka, tvořeného body vstupního patche, je rozdělena dvěma body na tři segmenty, kde sousední body jsou od sebe vzdáleny 1/3 délky hrany, na které leží. U každého z vytvořených bodů je provedena jeho projekce do roviny, která je kolmá na normálový vektor nejbližšího bodu vstupního patche. Střední bod je určen aritmetickým průměrem pozic všech vytvořených bodů a posunem, který je dán poloviční velikostí vektoru, určeným rozdílem zprůměrované pozice a pozice v rovině trojúhelníka z bodů vstupního patche.

$$
\begin{aligned}
w_{ij} &= (P_j - P_i) \cdot N_i \qquad (\cdot \text{ značí skalární součin}) \qquad (10.1)\\
B_{300} &= P_1\\
B_{030} &= P_2\\
B_{003} &= P_3\\
B_{210} &= (2P_1 + P_2 - w_{12}N_1)/3\\
B_{120} &= (2P_2 + P_1 - w_{21}N_2)/3\\
B_{021} &= (2P_2 + P_3 - w_{23}N_2)/3\\
B_{012} &= (2P_3 + P_2 - w_{32}N_3)/3\\
B_{102} &= (2P_3 + P_1 - w_{31}N_3)/3\\
B_{201} &= (2P_1 + P_3 - w_{13}N_1)/3
\end{aligned}
$$

$$
\begin{aligned}
E &= (B_{210} + B_{120} + B_{021} + B_{012} + B_{102} + B_{201})/6\\
V &= (P_1 + P_2 + P_3)/3\\
B_{111} &= E + (E - V)/2
\end{aligned}
$$

P_1, P_2, P_3 - *body původního trojúhelníka*
N_1, N_2, N_3 - *normálové vektory bodů původního trojúhelníka*

Výpočet jednotlivých bodů Beziérova trojúhelníka [14]



Obrázek 10.2: Znázornění způsobu projekce bodů do příslušné roviny

Výstupem TCS není jen výstupní patch, ale i hodnoty pro teselační level. V této implementaci je teselační level měněn na základě vzdálenosti objektu od pozorovatele, nicméně je nutné, aby byl specifikován stejný teselační level pro všechny plochy objektu z důvodu eliminace vzniku možných nespojitostí mezi plochami objektu. Teselační level je nastaven pomocí built-in proměnných *gl_TessLevelOuter* a *gl_TessLevelInner* [2]. Následující kódy obsahují funkce pro výpočet jednotlivých bodů Beziérova trojúhelníka a nastavení hodnot teselačního levelu.

```
//TCS_WorldPos[] is input patch

vec3 CalcPoint(uint n1,uint n2,uint n3,vec3 dirToPoint,vec3 normal)
{
    vec3 planePosition = n1 * TCS_WorldPos[0] + n2 * TCS_WorldPos[1] +
                         n3 * TCS_WorldPos[2];
    vec3 projectionVec = dot(dirToPoint, normal)  * normal;
    return (planePosition - projectionVec) / 3;
}

vec3 CalcMidPoint(Patch p)
{
    vec3 planePosition = (TCS_WorldPos[0] + TCS_WorldPos[1] +
                          TCS_WorldPos[2]) / 3;
    vec3 avgPosition = (p.B102 + p.B201 + p.B210 + p.B120 + p.B021 +
                        p.B012) / 6;
    return avgPosition + (avgPosition - planePosition) / 2;
}
```

```
float tessLevel = 10 - distance(objectPosition, cameraPosition);
tessLevel = clamp(tessLevel, 1, 10);

for(uint i = 0; i < 3; i++)
{
    gl_TessLevelOuter[i] = tessLevel;
}
gl_TessLevelInner[0] = tessLevel;
```

Fáze PG provádí teselaci nad trojúhelníkovou doménou, což znamená, že jsou vygenerovaným vrcholům přiřazeny barycentrické souřadnice, které jsou v TES přístupné pomocí built-in proměnné *gl_TessCoord* [2]. Tyto souřadnice jsou použity jako parametry pro výpočet bodu Beziérovy plochy a také pro výpočet normálového vektoru a texturovacích souřadnic pomocí lineární interpolace.

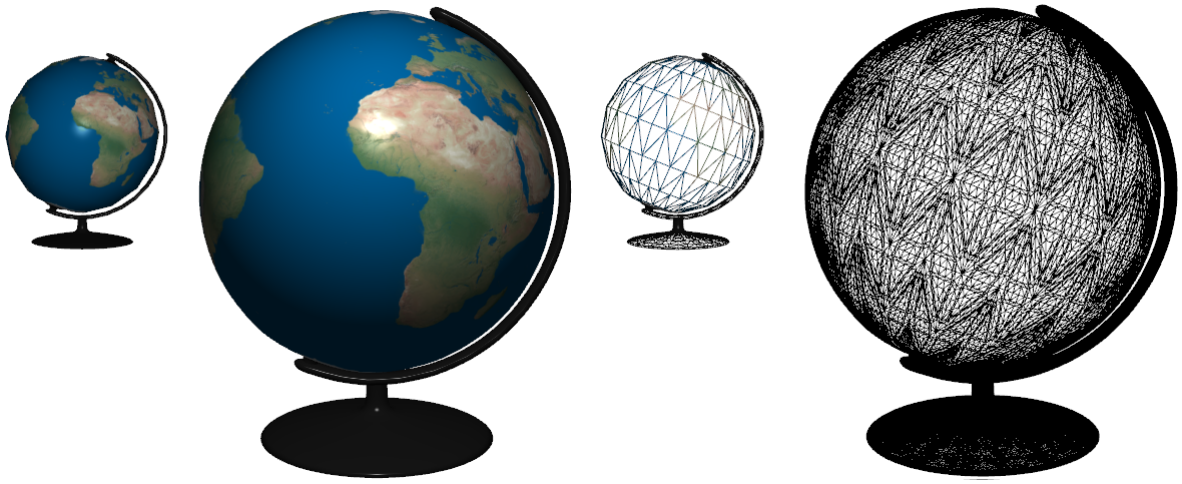
$$
\begin{aligned}
B(u,v) &= \sum_{i+j+k=3} B_{ijk} \frac{3!}{i!j!k!} u^i v^j w^k \\
&= B_{300}w^3 + B_{030}u^3 + B_{003}v^3 + \\
&B_{210}3w^2u + B_{120}3wu^2 + B_{201}3w^2v + \\
&B_{021}3u^2v + B_{102}3wv^2 + B_{012}3uv^2 + B_{111}6wuv
\end{aligned}
\tag{10.2}
$$

u, v, w - *parametry pro výpočet bodu, kde* $w = 1 - u \ - v$

Vzorec pro výpočet bodu Beziérova povrchu [14]

Následující kód představuje část hlavní funkce TES, kde dochází k výpočtu pozice a dalších vlastností vrcholu.

```
//TES_Patch is TCS output patch
//FS_WorldPos, FS_Normal, FS_UV are output attributes
void main(void)
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    float w = gl_TessCoord.z;

    float u2 = pow(u,2); //v2, w2 same principle
    float u3 = pow(u,3); //v3, w3 same principle

    FS_WorldPos =
        TES_Patch.B300 * w3 + TES_Patch.B030 * u3 +
        TES_Patch.B003 * v3 + TES_Patch.B210 * 3.0 * w2 * u +
        TES_Patch.B120 * 3.0 * w * u2 + TES_Patch.B201 * 3.0 * w2 * v +
        TES_Patch.B021 * 3.0 * u2 * v + TES_Patch.B102 * 3.0 * w * v2 +
        TES_Patch.B012 * 3.0 * u * v2 + TES_Patch.B111 * 6.0 * u * v * w;

    FS_Normal = u * TES_Patch.normal[0] + v * TES_Patch.normal[1] +
                w * TES_Patch.normal[2];
    FS_UV = u * TES_Patch.uv[0] + v * TES_Patch.uv[1] +
            w * TES_Patch.uv[2];

    gl_Position = projectionMatrix * viewMatrix * vec4(FS_WorldPos, 1.0);
}
```



Obrázek 10.3: Výsledek teselace s využitím Beziérova trojúhelníka

Výsledný obrázek znázorňuje vliv teselačního levelu na výsledné zaoblení objektu. Teselační level je měněn vzhledem ke vzdálenosti objektu od pozorovatele, kde s nižší hodnotou vzdálenosti je teselační level vyšší, což způsobuje generování více vrcholů, formujících Beziérův povrch.

Spustitelná ukázková aplikace pro teselaci je obsažena v příloze, ve složce /Examples/10. Tessellation .

Kapitola 11

Závěr

V této práci byly probrány vybrané možnosti využití moderního OpenGL, ať už se jedná o využití samostatných OpenGL funkcionalit nebo využití pro techniky, které jsou používány v počítačové grafice. Ke každé vybrané možnosti byl vytvořen jeden nebo více ukázkových příkladů, které demonstrují její praktické použití. Ukázkové příklady jsou vytvořeny tak, že obsahují pouze danou problematiku, případně další části pro funkčnost aplikace. Celkově bylo vytvořeno devět ukázkových aplikací, s cílem následného použití v předmětu ZPG nebo dalších podobných předmětech, zaměřených na OpenGL. Během práce byl také vytvořen dodatečný dokument, který obsahuje podrobnější informace o vybraných možnostech a který je součástí této práce, v podobě elektronické přílohy.

Pro určité techniky, které jsou obsahem této práce, je zřejmé, že budou v budoucnu nahrazeny technikami vyspělejšími a složitějšími. Jedná se například o výpočet osvětlení pomocí empirických osvětlovacích modelů nebo výpočet stínů, které mohou být časem nahrazeny výpočtem globálních osvětlovacích metod i v real-time aplikacích. Tato možnost vychází z rychlého vývoje a neustálého zvyšování výkonu grafických karet. Z tohoto důvodu je jednou z možností pokračování této práce, kterou bych se rád zabýval, zaměření na tyto metody.

Další možností pokračování je zaměření na komplexnější techniky s využitím standardu OpenGL, například na různé techniky vizualizace objemových dat, kde jedna z možných technik byla implementována i v rámci této práce. Vizualizace objemových dat je hojně využívána v oblasti lékařství, a proto je toto téma aktuální.

Seznam použité literatury

[1] SHERROD, Allen a Wendy JONES. *Beginning DirectX 11 game programming.* Boston, Mass.: Course Technology, c2012, xii, 372 pages ISBN 14-354-5895-8.

[2] *OpenGL - The Industry Standard for High Performance Graphics* [online]. ©1997-2014 [cit. 2014-03-19].
Dostupné z: http://www.opengl.org/

[3] SHREINER, Dave, Graham SELLERS, John M KESSENICH a Bill LICEA-KANE. *OpenGL programming guide: the official guide to learning OpenGL, version 4.3.* Eighth edition. United States: Addison-Wesley, 2013, xlvi, 935 pages. ISBN 03-217-7303-9.

[4] *Open Asset Import Library* [online]. © 2007-2009 [cit. 2014-03-19].
Dostupné z: http://assimp.sourceforge.net/

[5] *OpenCV* [online]. © 2014 [cit. 2014-03-19].
Dostupné z: http://opencv.org/

[6] CORRAL, Michael. *Vector Calculus.* 2013, 222 s.
Dostupné z: http://mecmath.net/calc3book.pdf

[7] ŽÁRA, Jiří, Bedřich BENEŠ, Jiří SOCHOR a Petr FELKEL. *Moderní počítačová grafika.* Vyd 1. Brno: Computer Press, 2004, 609 s. ISBN 80-251-0454-0.

[8] LENGYEL, Eric. *Computing Tangent Space Basis Vectors for an Arbitrary Mesh.* Terathon Software 3D Graphics Library [online]. 2001, č. 1 [cit. 2014-03-19].
Dostupné z: http://www.terathon.com/code/tangent.html

[9] KILGARD, Mark. *Shadow Mapping with Today's OpenGL Hardware.* In: NVIDIA Corporation [online]. 2001 [cit. 2014-03-19].
Dostupné z: http://www.slideshare.net/Mark_Kilgard/shadow-mappingwith-todays-opengl-hardware

[10] KILGARD, Mark. *Real-time Shadowing Techniques: Shadow Volumes.* In: NVIDIA Corporation [online]. 2003 [cit. 2014-03-19].
Dostupné z: http://www.slideshare.net/Mark_Kilgard/realtime-shadowing-techniques-shadow-volumes

[11] *Quaterniony.* K3dEngine - OpenGL Game Engine [online]. [2007] [cit. 2014-03-19].
Dostupné z: http://kengine.sourceforge.net/tutorial/vc/quaternion.htm

[12] *A faster quaternion-vector multiplication.* Molecular Musings [online]. 24.5.2013 [cit. 2014-03-19].
Dostupné z: http://molecularmusings.wordpress.com/2013/05/24/a-faster-quaternion-vector-multiplication/

[13] MARTIN, Brian. *Quaternion interpolation.* Theory Org [online]. 23.6.1999 [cit. 2014-03-19].
Dostupné z: https://theory.org/software/qfa/writeup/node12.html

[14] VLACHOS, Alex, Jörg PETERS, Chas BOYD a Jason L. MITCHELL. Curved PN triangles. *Proceedings of the 2001 symposium on Interactive 3D graphics - SI3D '01* [online]. New York, New York, USA: ACM Press, 2001, s. 159-166 [cit. 2014-03-30]. DOI: 10.1145/364338.364387.
Dostupné z: http://alex.vlachos.com/graphics/CurvedPNTriangles.pdf

[15] MEIRI, Etay. *OGL dev Modern OpenGL Tutorials* [online]. 19.10.2010 [cit. 2014-03-19].
Dostupné z: http://ogldev.atspace.co.uk

Seznam příloh

I. Elektronické dokumenty - Příloha na CD

i.	Elektronická verze práce	\Documents \Bakalářská práce
ii.	Teoretická část práce	\Documents \Bakalářská práce - teoretická část

II. Ukázkové příklady - Příloha na CD

i.	Objekty a textury	\Examples \01. - 02. Objects and textures \Examples \03. Volume rendering
ii.	Osvětlení	\Examples \04. Lighting
iii.	Normal mapping	\Examples \05. Normal mapping
iv.	Stíny	\Examples \06. Shadow mapping - point light \Examples \07. Stencil shadow volume - point light
v.	Animace	\Examples \08. Animation
vi.	Částicový systém	\Examples \09. Particle system - fire
vii.	Teselace	\Examples \10. Tessellation